# CYBIL for NOS/VE Language Definition

## Usage

Publication Number 60464113

# Manual History

| Revision | System Version | PSR Level | Date |
|---|---|---|---|
| A | 1.0.2 | - | February 1984 |
| B | 1.1.1 | 613 | July 1984 |
| C | 1.1.2 | 630 | March 1985 |
| D | 1.1.3 | 644 | October 1985 |
| E | 1.2.1 | 664 | September 1986 |
| F | 1.2.2 | 678 | April 1987 |

This manual is Revision F, printed in April 1987. It reflects NOS/VE Version 1.2.2 at PSR level 678.

New features:

- The Common CYBIL Input/Output procedures (Part II of this manual)
- Two new appendixes: Appendix G briefly describes how to use NOS/VE commands and utilities, and how to get online information; Appendix L explains how to access the online examples.

Other changes:

- Chapter 9 is now a brief, step-by-step introduction to using the Debug utility. The detailed descriptions of the subcommands and functions of the Debug Utility have been moved to the Debug Usage manual, publication number 60488213.

Miscellaneous technical corrections and clarifications have been incorporated, together with editorial changes. This edition obsoletes all previous editions.

# Contents

## Figures

## Tables

# About This Manual

This manual describes CONTROL DATA® CYBIL and CYBIL system-resident procedures. CYBIL is the implementation language of CDC® Network Operating System/Virtual Environment (NOS/VE).

NOS/VE provides a program interface consisting of a large number of CYBIL procedures. Each CYBIL procedure supplies a specific system service to CYBIL programs. Descriptions of the CYBIL procedures are topically divided and appear in several manuals, of which this manual is one.

This manual defines the CYBIL language in detail and describes the CYBIL Input/Output procedures for reading and writing files and for other I/O-related functions on CYBIL programs.

## Audience

This manual is written for CYBIL programmers. It assumes that you understand NOS/VE and System Command Language (SCL) concepts as presented in the SCL System Interface and SCL Language Definition manuals.

## The CYBIL Manual Set

This manual is part of a set of manuals describing CYBIL. Descriptions of all manuals in the CYBIL manual set follow:

**CYBIL Language Definition**

Defines the CYBIL language in detail and describes the CYBIL Input/Output procedures for reading and writing files and for other I/O-related functions on CYBIL programs.

**CYBIL System Interface**

Describes the CYBIL procedures that pertain to command language services and processing, program services and management, task and job management services, condition processing, message generation, interstate communication, limits, and statistics.

**CYBIL File Management**

Describes the CYBIL procedures that assign files to device classes, specify attributes for files, and perform file opening, closing, and copying.

**CYBIL Sequential and Byte-Addressable Files**

Describes segment and record access, and input/ouput operations to sequential and byte-addressable files.

**CYBIL Keyed-File and Sort/Merge Interfaces**

Describes the following:

- The interface to NOS/VE keyed-files (files having the indexed-sequential and direct-access file organizations).
- The interface to NOS/VE Sort/Merge, which is used to sort records or merge files of sorted records.

## Organization of This Manual

This manual is in two parts:

- Part I, which describes the CYBIL language, is organized by topic, based on elements of the language. The first chapter introduces the basic elements of the language and refers to the chapter in which each element is described.
- Part II explains how to use CYBIL I/O procedures and describes each procedure.

## Conventions

The following conventions are used in this manual:

| | |
|---|---|
| **boldface** | In a command or function format, names and required parameters are in boldface type. |
| *{name}* | Optional parameters are shown in italics and are enclosed by braces. If the parameter is optional and can be repeated any number of times, it is also followed by several periods: *{name}...* Braces also indicate that the enclosed parameters and reserved words are used together. For example: *{offset MOD base}* is considered a single parameter. Except for the braces and periods indicating repetition, all other symbols shown in the format must be included in the coding. |
| UPPERCASE | Uppercase is used for names of commands, functions, and parameters (and their abbreviations). Uppercase is also used for names of variables, files, system constants, and terminal keys and function keys when they occur in text. |
| numbers | All numbers are decimal unless otherwise noted. |
| return | Represents the message transmission key on your terminal. Depending on the terminal, this key may be the RETURN, NEXT, CR, CARRIAGE RETURN, NEW LINE, SEND, or ETX key. |
| shift | Represents the shift key on your terminal. |
| vertical bar | A technical change is indicated by a vertical bar next to the change. |

| | |
|---|---|
| `examples` | Examples are in lowercase unless uppercase characters are required for accuracy. Interactive terminal session examples are shown in a type font that resembles computer output. |
| blue | Within interactive terminal sessions, user input is printed in blue; system output is printed in black. |

## Submitting Comments

There is a comment sheet at the back of this manual. You can use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report errors. Mail your comments to:

Control Data Corporation
Technology and Publications Division ARH219
4201 North Lexington Avenue
St. Paul, Minnesota 55126-6198

Please indicate whether you would like a response.

If you have access to SOLVER, the Control Data online facility for reporting problems, you can use it to submit comments about the manual. When entering your comments, use CIL as the product identifier. Include the name and publication number of the manual.

## In Case of Trouble

Control Data's Central Software Support maintains a hotline to assist you if you have trouble using our products. If you need help not provided in the documentation, or find the product does not perform as described, call us at one of the following numbers. A support analyst will work with you.

From the USA and Canada: (800)-343-9903

From other countries: 612-851-4131

If you have questions about the packaging and/or distribution of a printed manual, write to:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Or call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

# Part I. The CYBIL Language

# Introduction 1

This chapter introduces the basic elements of a CYBIL program and refers you to the chapter in which each is further described.

# Introduction 1

A CYBIL program consists of two kinds of elements: declarations and statements. Declarations describe the data to be used in the program. Statements describe the actions to be performed on the data.

Declarations and statements are made up of predefined reserved words and user-defined names and values. The way you form these elements is described in chapter 2, as is the general structure for designing a CYBIL program.

## Declarations

Data can be either constant or variable. You can use the constant value itself or give it a name using the constant declaration (CONST). Variables are named, initialized, and given certain characteristics with the variable declaration (VAR).

One of the characteristics of a variable is its type, such as integer or character. You can use CYBIL's predefined (standard) types or define your own types.

To define a new type or redefine an existing type with a new name, use the type declaration (TYPE). Once you have defined a type, CYBIL will treat it as a standard data type; If you specify your new type name as a valid type in a variable declaration, CYBIL will perform standard type checking on it. You can also declare where you want certain variables to reside by defining an area called a section, which can be a read-only section or a read/write section. This is done with the SECTION declaration. All of these data-related declarations are described in chapter 3.

Many standard types are available, among which integers, floating-point numbers, characters, and boolean values. In addition, you can use combinations of the standard types to define your own data types, for example, a record that contains several fields. The next paragraphs summarize the types that are predefined by CYBIL. They are described in detail in chapter 4.

The standard types can be grouped into three categories: basic, structured, and storage.

The *basic types* are:

- *Scalar,* which are typess that have a specific order. They include integer, character, boolean, ordinal (in which you define the elements and their order), and subrange (which can be specified for any of the scalar types by giving a lower and upper bound).

- *Floating-point* (real).

- *Cell,* which represents the smallest addressable unit of memory.

- *Pointer,* which points to a variable, allowing you to access the variable by location rather than by name.

With these basic types you can construct the *structured types:* strings, arrays, records, and sets.

- *String* is a sequence of characters. You can reference a portion of a string (called a substring) or a single character within a string.

- *Array* is a structure that contains components all of the same type. The components of an array have a specific order and each one can be referenced individually.

- *Record* is a structure that contains a fixed number of fields, which may be of different types. Each field has a unique name within the record and can be referenced individually. You can also declare a variant record that has several possible variations (variants). The current value of a field common to all variants, or the latest assignment to a specific variant field determines which of the variants should be used for each execution.

- *Set* is a structure that contains elements of a single type. Unlike an array, elements in a set have no order and individual elements cannot be referenced. A set can be operated on only as a whole.

*Storage types* are structures to which variables can be added, referenced, and deleted under explicit program control using a set of storage management statements. The two storage types are *sequence* and *heap.*

All of the types mentioned above are considered fixed types; that is, there is a definite size associated with each one when it is declared. If you want to delay specifying a size until execution time, you can declare it as an adaptable type. Then, sometime during execution, you assign a fixed size or value to the type. A string, array, record, sequence, or heap can be adaptable.

All of these types are described in chapter 4.

## Statements

Statements define the actions to be performed on the data you have defined. There are four kinds of statements:

- The *assignment statements* change the value of a variable.
- *Structured statements* contain and control the execution of a list of statements: The BEGIN statement unconditionally executes a statement list. The WHILE, FOR, and REPEAT statements control repetitive executions of a statement list.
- *Control statements* control the flow of execution. The IF and CASE statements execute one of a set of statement lists based on the evaluation of a given expression or the value of a specific variable. CYCLE, EXIT, and RETURN statements stop execution of a statement list and transfer control to another place in the program.
- *Storage management statements* allocate, access, and release variables in sequences (using the RESET and NEXT statements), heaps (using the RESET, ALLOCATE, and FREE statements), and the run-time stack (using the PUSH statement).

All of the preceding statements are described in detail in chapter 5, along with the operands and operators that can be used in expressions within statements and declarations.

Statements can appear within a program (as described in chapter 2), a function, or a procedure.

A *function* is a list of statements, optionally preceded by a list of declarations. It is known by a unique name and can be called by that name from elsewhere in the program. A function performs some calculation and returns a value that takes the place of the function reference. There are many standard functions defined in CYBIL and you can also create your own. Standard functions and rules for forming your own functions are described in chapter 6.

A *procedure*, like a function, is a list of statements, optionally preceded by a list of declarations. It also is known by a unique name and can be called by that name from elsewhere in the program. A procedure performs specific operations and may or may not return values to existing variables. You can use the standard procedures or define your own. Chapter 7 describes the standard procedures and rules for forming your own procedures.

Chapter 8 describes how to compile and format CYBIL source code. The CYBIL command and directives embedded in the source code specify how compilation should be performed. The CYBIL command calls the CYBIL compiler, tells it which files to use for input and output, and specifies what kind of listing you want. The text-embedded compilation directives specify listing options, run-time options, the layout of the source text, and which portions of the source text to compile.

The FORMAT_CYBIL_SOURCE command and other text-embedded directives specify how formatting should be performed. For example, they indicate the margins and line width, tab settings, and indentation to be used in the program listing.

In summary, chapters 2 through 7 describe the elements within a CYBIL program. Chapter 8 describes the commands and directives that control how the program is compiled and formatted.

## CYBIL I/O Procedures

Procedures that perform input to and output from CYBIL programs are described in this manual in Part II, Common CYBIL Input/Output. Other procedures that perform input and output on CYBIL programs are described in the CYBIL File Management manual, the CYBIL Sequential and Byte-Addressable Files manual, and the CYBIL Keyed-File and Sort/Merge Interfaces manual.

# Program Structure 2

This chapter describes how to form the individual elements used within a program and how to structure the program itself.

# Program Structure 2

This chapter describes how to form the individual elements used within a program and how to structure the program itself.

## Elements Within a Program

This section describes valid characters, CYBIL-defined elements, user-defined elements, and syntax.

### Valid Characters

The characters that can be used within a program are those in the ASCII character set that have graphic representations (that is, can be printed). This character set is included in appendix C. It contains uppercase and lowercase letters. In names that you define, you can use uppercase and lowercase letters interchangeably. For example, the name LOOP_COUNT is equivalent to the name loop_count.

### CYBIL-Defined Elements

CYBIL has predefined meanings for many words and symbols. You cannot redefine or use these words and symbols for other purposes.

A complete list of CYBIL reserved words is given in appendix D. In the formats for declarations, type specifications, and statements shown in this manual, reserved words are shown in uppercase letters.

The following list includes the reserved symbols and a brief description of the purpose of each. They are discussed in more detail throughout this manual.

| Symbol | Purpose |
|---|---|
| +, −, *, /, =, <, <=, >, >=, <>, :=, (,) | Operators used in expressions. They are discussed in chapter 5. |
| ; | Separates individual declarations and statements. |
| : | Used in declarations as described in chapter 3. |
| , | Separates repeated parameters or other elements. |
| . | Indicates a reference to a field within a record as described in chapter 4. |
| .. | Indicates a subrange as described in chapter 4. |
| ^ | Indicates a pointer reference as described in chapter 4. |
| ' ' | Delimits a string. |
| [ ] | Encloses array subscripts, indefinite value constructors, and set value constructors as described in chapter 4. |
| { } | Delimits comments. (Within the formats shown in this manual, they are also used to enclose optional parameters.) |
| ? or ?? | Indicates compile-time statements and directives as described in chapter 8. |

## User-Defined Elements

This section describes names, constants, and constant expressions.

### Names

You define the names for elements, such as constants, variables, types, procedures, and so on, that you use within a program. A name:

- Can be from 1 to 31 characters in length
- Can consist of letters, digits, and the special characters # (number sign), @ (commercial at sign), _ (underline), and $ (dollar sign) [1]
- Must begin with a letter (there is an exception to this rule for system-defined functions and procedures that begin with the # or $ character)
- Cannot contain spaces
- Cannot be a reserved word (a complete list of CYBIL reserved words is given in appendix D)

In the formats included in this manual, names that you supply are shown in lowercase letters. Within a program, however, there is no distinction between uppercase and lowercase letters. The name my_file is identical to the name My_File.

1. NOS/VE often uses $ in its predefined names. To keep from matching a system-reserved name, avoid using $ in the names you define.

There is considerable flexibility in forming names, so you should make them as descriptive as possible to promote readability and maintainability of the program. For example, LAST_FILE_ ACCESSED is more obvious than LASTFIL.

Examples:

| **Valid Names** | **Invalid Names** |
|---|---|
| SUM | ARRAY |
| REGISTER#3 | FILES&POSITIONS |
| POINTER_TABLE | 2ND |

The valid names need no explanation. Among the invalid names, ARRAY cannot be used because it is a reserved word; FILES&POSITIONS contains an invalid character (the ampersand); and 2ND does not begin with a letter.

## Constants

A constant is a fixed value. It is known at compilation time and does not change throughout the execution of a program. It can be an integer, character, boolean, ordinal, floating-point number, pointer, or string.

Integer constants can be binary, octal, decimal, or hexadecimal. The base is specified by enclosing the radix in parentheses following the integer, as follows:

    integer (radix)

Examples are 1011(2) and 19A(16). If the radix is omitted, the integer is assumed to be decimal. Integer constants must start with a digit; therefore, 0 must precede any hexadecimal constant that would otherwise begin with a letter, for example, 0FF(16). Negative integer constants must be preceded by a minus sign. Positive integer constants can be preceded by a plus sign but need not be.

Integer constants range in value from $-(2^{63}-1)$ to $2^{63}-1$; that is, -7FFFFFFFFFFFFFFF hexadecimal through 7FFFFFFFFFFFFFFF hexadecimal.

A character constant can be any single character in the ASCII character set. The character is enclosed in apostrophes in the following form:

    'character'

Examples are 'A' and '?'. The apostrophe character itself is specified by a pair of apostrophes.

A boolean constant can be either TRUE or FALSE, each having its usual meaning.

An ordinal constant is an element of an ordinal type that you have defined. As a defined element of an ordinal type, it is referred to as an ordinal name. For further information, refer to Ordinal under Scalar Types in chapter 4.

Floating-point (real) constants can be written in either decimal notation or scientific notation. A real number written in decimal notation contains a decimal point and at least one digit on each side, for example, 5.123 or -72.18. If the number is positive, the sign is optional; if negative, the sign is required.

A real number written in scientific notation is represented by a number (the coefficient), which is multiplied by a power of 10 (the exponent) in the form:

```
coefficientEexponent
```

The prefix E is read as "times 10 to the power of." For example,

```
5.1E6
```

is 5.1 times 10 to the power of 6, or 5,100,000. The decimal point in the coefficient is optional. A decimal point cannot appear in the exponent; it must be a whole number. If the coefficient or exponent is positive, the sign is optional; if negative, the sign is required.

The pointer constant is NIL. It indicates an unassigned pointer. For CYBIL on NOS/VE, a pointer is represented partially by an address called the process virtual address (PVA). The PVA is represented as a packed record consisting of three fields: the ring number, segment number, and byte offset. To indicate the NIL pointer constant internally, CYBIL sets these three fields to 0F hexadecimal, 0FFF hexadecimal, and 80000000 hexadecimal, respectively. NIL can be assigned to a pointer of any type.

String constants consist of one or more characters enclosed in apostrophes in the form:

```
'string'
```

An example is 'USER1234', a string of eight characters. An apostrophe in a string constant is specified by a pair of apostrophes, for example, 'DON''T'.

String constants can be concatenated by using the reserved word CAT, as in:

```
'characters_1' CAT 'characters_2'
```

The result is the string 'characters_1characters_2'. The CAT operation cannot be used with string variables.

A string constant can be empty, that is, a null string; for example,

```
CONST str = '';
```

declares the string constant STR to be a null string. As a result of this declaration, the length of STR is set to zero.

You cannot reference parts (substrings) of string constants.

## Constant Expressions

Expressions are combinations of operands and operators that are evaluated to find scalar or string type values. In a constant expression, the operands must be constants, names of constants (that you declare using the constant declaration described in chapter 3), or other constant expressions within parentheses. Computation is done at compilation time and the resulting value used in the same way a constant is used.

The general rules for forming and evaluating expressions are described under Expressions in chapter 5. These rules apply to constant expressions with the following exceptions:

- Constant expressions must be simple expressions; terms involving relational operators must be delimited with parentheses.
- The only functions allowed as factors in constant expressions are the $INTEGER, $CHAR, SUCC, and PRED functions with constant expressions as arguments.
- Substring references are not allowed.

## Syntax

The exact syntax of the language is shown in the formats of individual declarations and statements described in the remainder of this manual. The following paragraphs discuss general syntax rules.

### Spaces

Spaces can be used freely in programs with the following exceptions:

- Names and reserved words cannot contain embedded spaces. Normally, constants cannot contain spaces either, but a character constant or string constant can.
- A name, reserved word, or constant cannot be split over two lines; it must appear completely on one line.
- Names, reserved words, and constants must be separated from each other by at least one space, or by one of the other delimiters such as a parenthesis or comma.

For further information, refer to Spacing later in this chapter.

### Comments

Comments can be used in a program anywhere that spaces can be used (except in string constants). They are printed in the source listing but otherwise are ignored by the compiler.

A comment is enclosed in left and right braces. It can contain any character except the right brace. To extend a comment over several lines, repeat the left brace at the beginning of each line. If the right brace is omitted at the end of the comment, the compiler ends it automatically at the end of the line.

Example:

```
{this comment
{appears on
{several lines.}
```

Within this manual, the formats for declarations, type specifications, and statements use braces to indicate an optional parameter.

## Punctuation

A semicolon separates individual declarations and statements. It must be included at the end of almost every declaration and statement. The single exception is MODEND which can, but need not, end with a semicolon if it is the last occurrence of MODEND in a compilation. Punctuation for specific declarations and statements is shown in the formats in the following chapters.

Two consecutive semicolons indicate an empty statement, which the compiler ignores. Spacing between the semicolons in this case is unimportant.

## Spacing

Declarations and statements can start in any column. In this manual, indentations are used in examples to improve readability. It is recommended that similar conventions be used in your programs to aid in debugging and documentation for yourself and other users. The CYBIL source code formatter, described in chapter 8, can help you do this by accepting source code you supply as input and formatting it for consistency and readability.

The LEFT and RIGHT directives, described in chapter 8, can be used at compilation time to specify the left and right margins of the source text. All source text outside of those margins is then ignored. A warning diagnostic is issued for every line that exceeds the specified right margin.

A name, reserved word, or constant cannot be split over two lines; each must appear completely on one line.

## Structure of a Program

This section describes the module structure, scope, module declaration, and program declaration.

### Module Structure

The basic unit that can be compiled is a module and, optionally, compilation time statements and directives. A module can, but need not, contain a program. Use this general structure for a module:

```
MODULE module_name;
    declarations
    PROGRAM program_name;
        declarations
        statements
    PROCEND program_name;
MODEND module_name;
```

Declarations can be constant, type, variable, section, function, and procedure declarations. A module can contain any number and combination of declarations, but it can contain only one program. The program contains the code (that is, the statements) that are actually executed. The required module and program declarations are described later in this chapter.

The structure within a module determines the scope of the elements you declare within it.

### Scope

The scope of an element you declare, such as a variable, function, or procedure, is the area of code where you can refer to the element and it will be recognized. Scope is determined by the way the program and procedures are positioned in a module and where the elements are declared.

In terms of scope, the programs, procedures, and functions are often referred to as blocks (that is, blocks of code). If an element is declared within a block, its scope is only that block (unless it is externally declared as described later in this section). Outside the block, the element is unknown and references to it are not valid. A variable declared within a block is said to be local to the block and is called a local variable.

An element declared at the module level (that is, one that is not declared within a program, procedure, or function) has a scope of the entire module. It can be referred to anywhere within the module. A variable declared at the module level is said to be global and is called a global variable.

A block can contain one or more subordinate blocks. A variable declared in an outer block can always be referenced in a subordinate block. However, if a subordinate block declares an element of the same name, the new declaration applies while inside that block. Figure 2-1 illustrates these rules.

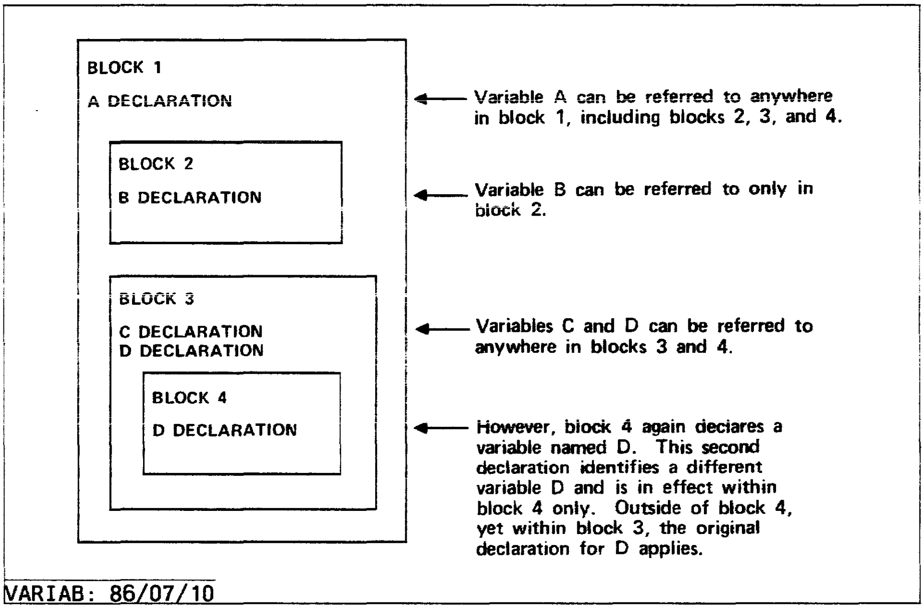

BLOCK 1
A DECLARATION
Variable A can be referred to anywhere in block 1, including blocks 2, 3, and 4.
BLOCK 2
B DECLARATION
Variable B can be referred to only in block 2.
BLOCK 3
C DECLARATION
D DECLARATION
Variables C and D can be referred to anywhere in blocks 3 and 4.
BLOCK 4
D DECLARATION
However, block 4 again declares a variable named D. This second declaration identifies a different variable D and is in effect within block 4 only. Outside of block 4, yet within block 3, the original declaration for D applies.
VARIAB: 86/07/10

**Figure 2-1. Scope of Variables Within a Block Structure**

Storage space is allocated for a variable when the block in which it is declared is entered. Space is released when an exit is made from the block. Because space is allocated and released automatically, these variables are called automatic variables. This method of allocation becomes more complex when a procedure, for example, calls itself. Space for the same variable must be allocated each time the procedure is called and entered, yet each of these spaces must be kept separate to maintain the integrity of the variable throughout each execution of the procedure. More discussion about this recursive process appears later in this section.

You can specify that storage for a variable remains intact throughout execution by including the STATIC attribute when you declare the variable. A variable declared in this way is called a static variable.

A global variable is always static. Because it is declared at the outermost level of a module (consider the module to be a block), storage for a global variable is allocated throughout execution of the module (or block). For further information on automatic and static variables, refer to Variable Declaration in chapter 3.

Storage is allocated dynamically when the number of times a variable will occur is unknown. This happens in two cases, the first of which is in recursive procedures or functions when the number of recursive calls is unknown. In this case, a mechanism called a stack frame is used to hold the automatic variables. Stack frames are allocated automatically and you need not understand them to use recursive procedures and functions. However, a more complete description is given in appendix F, The CYBIL Run-Time Environment.

The second case of dynamic storage allocation is in the use of the special storage types heaps and sequences (described in chapter 4) and the run-time stack (described in chapter 5). Heaps and sequences represent structures to which you can add and delete variables under program control. To allocate space in a heap, you use the ALLOCATE statement; in a sequence, you use the NEXT statement. The run-time stack is a structure to which you can add but not explicitly delete variables. To allocate space on the run-time stack, you use the PUSH statement. Space is released when the procedure containing the PUSH statement completes. All three statements are described in chapter 5 under Storage Management Statements.

The one exception to the preceding scope rules is an element declared with the XDCL (externally declared) attribute. This attribute means the element is declared in one module but can be referred to in another. In this case, the loader handles the links between modules. For further information on the XDCL attribute, refer to chapter 3.

## Module Declaration

The module declaration marks the beginning of a module. MODEND marks the end of a module. A module can contain at most one program declaration and any combination of type, constant, variable, section, function, and procedure declarations. If two or more modules are compiled and linked together for execution, there can be only one program declaration in all the linked modules.

Use this format for a module declaration:

**MODULE name;**[2]

**name**

The name of the module.

Use this format for MODEND:

**MODEND** *{ name }* ;

*name*

The name of the module. This parameter is optional. If used, the name must be the same as that specified in the module declaration.

When compiling more than one module, a semicolon is required after each occurrence of MODEND except the last one (there it is not required, but is recommended).

2. Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a module declaration. If it is included in a CYBIL program run on NOS/VE, this parameter is ignored.

Examples:

The following example shows a module named ONE that contains various declarations and a program named MAIN.

```
MODULE one;

  declarations

  PROGRAM main;

    declarations

    statements
  PROCEND main;
MODEND one;
```

The following example shows a compilation consisting of three modules named ONE, TWO, and THREE. All three modules can be compiled and the resulting object modules linked together to form a single object module that can then be executed. For readability, the module names are included in all occurrences of MODEND.

```
MODULE one;

  declarations/statements
MODEND one;
MODULE two;

  declarations/statements
MODEND two;
MODULE three;

  declarations/statements
MODEND three;
```

## Program Declaration

The program declaration marks the beginning of a program. The end of a program is marked by a PROCEND statement. A program can contain any combination of type, constant, variable, section, function, and procedure declarations, and any statements. If two or more modules are compiled and linked together for execution, there can be only one program declaration in the linked modules.

Use this format for a program declaration:

**PROGRAM name** *{(formal_parameters)}* ;[3]

**name**

The name of the program.

*formal_parameters*

One or more optional parameters included if the program is to be called by the operating system. They can be in the form

**VAR name** *{,name}...* **: type**
*{,name {,name}... : type}...*

and/or

**name** *{,name}...* **: type**
*{,name {,name}... : type}...*

where **name** is the name of the parameter and **type** is the type of the parameter, that is, a predefined type (described in chapter 4) or a user-defined type (described in chapter 3).

The first form is called a reference parameter; its value can be changed during execution of the program. The second form is called a value parameter; its value cannot be changed by the program. Both kinds of parameters can appear in the formal parameter list; if so, they must be separated by semicolons (for example, I:INTEGER; VAR A:CHAR).

3. Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a program declaration. If it is included in a CYBIL program run on NOS/VE, this parameter is ignored.

The optional formal parameter list is included if a CYBIL program is to be called by the operating system. It allows the system to pass values (for example, a string that represents a command) to a CYBIL program. For further information on passing parameters, refer to the CYBIL System Interface manual.

Use this format for PROCEND:

**PROCEND** *{ name }* **;**

*name*

The name of the program. This parameter is optional. If used, the name must be the same as that specified in the program declaration.

Example:

The following example shows a program named MAIN that contains various declarations, including a procedure named SUB_1:

```
PROGRAM main;

  declarations

  PROCEDURE sub_1;

    declarations

    statements
  PROCEND sub_1;
  statements    {The program starts execution here.}
PROCEND main;
```

# Constant, Variable, Type, and Section Declarations 3

This chapter describes how you declare constant and variable data types and new data types. It also describes how you specify a particular section in which to group data.

# Constant, Variable, Type, and Section Declarations 3

This chapter describes the constant declaration, which defines a name for a value that never changes; the variable declaration, which defines a name for a value that can change; and the type declaration, which defines a new type of data and gives a name to that type. In addition, it also describes the section declaration, which groups variables that share common access characteristics.

## Constant Declaration

A constant, as described in chapter 2, is a fixed value that is known at compile time and doesn't change during execution. A constant declaration allows you to associate a name with a value and use that name instead of the actual constant value. This provides greater readability because the name can be descriptive of the constant. Constant declarations also provide greater maintainability because the constant value need only be changed in one place, the constant declaration, not every place it is used in the code.

Use this format for a constant declaration:

**CONST name = value** *{,name = value}...*;

**name**

The name associated with the constant value.

**value**

The constant value. It can be an integer, character, boolean, ordinal, floating-point, pointer, string, or constant expression. Rules for forming these values are given under Constants and under Constant Expressions in chapter 2.

You can write several constant declarations, each declaring a single constant, or a single declaration declaring several constants where each **name = value** combination is separated by a comma.

Type is not specified in a constant declaration. The type of the constant is the same as the type of the value assigned to it.

If used, an expression is evaluated during compilation. The expression itself can contain other constants.

Examples:

Rather than repeat the value of pi throughout a program, you can use a constant declaration to assign a descriptive name (in this case, PI) to the value and use that name in subsequent expressions and operations. The constant declaration is:

```
CONST
  pi = 3.1415927;
```

The following example shows a constant declaration containing several different types:

```
CONST
  first = 1,
  last = 80,
  hex = 0a8(16),
  bit_pattern = 10110101(2),
  fp_number = 1.2e3,
  stop_character = '.',
  continue = TRUE,
  message = 'end of line',
  last_pointer = NIL,
  length = last - first,
  result = (1 * 2) DIV 3;
```

Each constant has the same type as the value assigned to it. For example, FIRST and LAST are integer types, as is LENGTH, which is the result of an expression containing integers. Notice that the value of HEX begins with a 0 because integers must begin with a digit.

## Variable Declaration

A variable is an element within a program whose value can change during execution. The name of the variable stays the same; it is only the value contained in the variable that changes. To use a variable, you must declare it.

Use this format for a variable declaration:

> **VAR name** *{,name}...* **:** *{[attributes]}* **type** *{:= initial_value}*
> *{,name {,name}... :{[attributes]} type {:= initial_value}}...***;**[1]

**name**

The name of the variable. Specifying more than one name indicates that all of the named variables will have the characteristics that follow (attributes, type, and initial_value).

*attributes*

One or more of the following attributes. If you specify more than one, separate them with commas.

READ

Access attribute specifying that the variable is a read-only variable; the compiler checks to ensure that the value of the variable is not changed. If you specify READ, you must also specify an initial value.

XDCL

Scope attribute specifying that the variable is declared in this module but can be referenced from another module.

XREF

Scope attribute specifying that the variable is declared in another module but can be referenced from this module.

1. Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a variable declaration. If it is included in a CYBIL program run on NOS/VE, this parameter is ignored.

#GATE[2]

This attribute is undefined for variable declarations. However, if you specify #GATE, you must also specify the XDCL attribute.

STATIC

Storage attribute specifying that storage space for the variable is allocated at load time and remains when control exits from the block. Static storage is assumed when any attributes are specified.

section_name

Storage attribute specifying the name of the user-defined section in which the variable resides. A variable in a section that is defined as read-only is protected by hardware, as opposed to software. The section name and its read/write attributes must be declared using the section declaration (discussed later in this chapter).

Attributes are described in more detail later in this chapter.

The attributes parameter is optional. If it is omitted, CYBIL assumes the variable can be read and written; can be referenced only within the block where it is created; and, unless it is declared at the outermost level of a module, is automatic (that is, storage for the variable is allocated only during execution of the block in which the variable is declared).

**type**

Data type defining the values that the variable can have. Only values within this data type are allowed. Types are described in chapter 4.

*initial_value*

Initial value assigned to the variable. Specify a constant expression, an indefinite value constructor (described under Initialization later in this chapter), or a pointer to a global procedure. Only a static variable can be assigned an initial value. Initialization is discussed later in this chapter.

This parameter is optional. If it is omitted, the variable is undefined and filled with the loader's preset value.

2. This attribute is not supported on variations of CYBIL available on other operating systems.

Any variable referenced in a program must be declared with the VAR declaration. A variable can be declared only once at each block level although it can be redefined in another block or in a contained (nested) block.

The type assigned to a variable defines the range of values it can take on and also the operations, functions, and procedures that can use it. CYBIL checks to ensure that the operations performed on variables are compatible with their types.

Examples:

The following declarations define a variable named SCORES that can be any integer number, a variable named STATUS that can be either of the boolean values FALSE or TRUE, and two variables named ALPHA1 and ALPHA2 that can be characters:

```
VAR
  scores: integer;

VAR
  status: boolean;

VAR
  alpha1: char;

VAR
  alpha2: char;
```

The declarations for the two character type variables, ALPHA1 and ALPHA2, could be combined as follows:

```
VAR
  alpha1,
  alpha2: char;
```

To combine all of the variables in one declaration, you could use:

```
VAR
  scores: integer,
  status: boolean,
  alpha1,
  alpha2: char;
```

## Attributes

Attributes control three characteristics of a variable:

| Attribute | Characteristic |
|---|---|
| Access | Whether the variable can be both read and written |
| Scope | Where within the program the variable can be referenced |
| Storage | When and where the variable is stored |

### Access

The access attribute that you can specify is READ. A variable declared with the READ attribute can only be read. It must be initialized in the declaration and cannot be assigned another value later. It is called a read-only variable. If the READ attribute is omitted, CYBIL assumes the variable can be both read and written (changed).

The READ attribute is enforced by software; that is, the compiler checks to ensure that the value of a variable does not change. The READ attribute alone does not mean that the variable is actually in a read-only section.[3] To do that, you must specify the name of a read-only section as declared in a section declaration (described later in this chapter).

A variable with the READ attribute specified is assumed to be static. (For further information on static variables, refer to Storage later in this chapter.) You can use a read-only variable as an actual parameter in a procedure call only if the corresponding formal parameter is a value parameter; that is, a read-only variable can be passed to a procedure only if the procedure makes no attempt to assign a value to it. (Procedure parameters are described in chapter 7.)

3. A read-only section is a hardware feature. Data that resides in a physical area of the machine designated as a read-only section is protected by hardware, not by software. This feature is described in further detail in volume II of the virtual state hardware reference manual.

A read-only variable is similar to a constant, but can't always be used in the same places. For example, the initial value that you can assign to a variable (as described earlier in this chapter) must be a constant expression, an indefinite value constructor, or a pointer to a global procedure. In this case, even though a read-only variable has a constant value, you cannot use it in place of a constant expression. Also, as mentioned in chapter 2, you cannot reference a substring of a constant. You can, however, reference a substring of a variable and, thus, a read-only variable. There are other differences similar to these. The descriptions in this manual state explicitly whether constants and/or variables can be used.

Examples:

In this example the variable DEBUG is a read-only variable set to the constant value of TRUE. NUMBER can be read and written.

```
VAR
  debug: [READ] boolean := TRUE,
  number: integer;
```

The following example illustrates a difference between constants and read-only variables. To declare a string type, you must specify the length of the string in parentheses following its name. As defined in chapter 4, the length must be a positive, integer constant expression.

```
CONST
  string_size_1 = 5;

VAR
  string_size_2: [READ] integer := 5,
  string1: string (string_size_1),
  string2: string (string_size_2);
```

The declaration of STRING1 is valid; the length of the string is 5, which is the value of the constant STRING_SIZE_1. However, STRING2 is invalid; even though STRING_SIZE_2 does not change in value, it is still a variable and cannot be used in place of a constant expression.

## Scope

The scope attributes define the part or parts of a module to which a variable declaration applies. If you don't include any scope attributes in the declaration, the scope of a variable is the block in which it is declared. A variable declared in an outermost block applies to that block and all the blocks it contains. However, a variable declared even at the outermost level of a module cannot be used outside of that module. Use the scope attributes, XDCL and XREF, to extend the scope of a variable so that it can be shared among modules.

To use the same variable in different modules, you must specify the XDCL and XREF attributes. The XDCL attribute indicates that the variable being declared can be referenced from other modules. The XREF attribute indicates that the variable is declared in another module. When the loader loads modules, it resolves variable declarations so that each XDCL variable is allocated static storage and the XREF variable shares the same space. This is known as satisfying externals. The loader issues an error if an XREF variable does not have a corresponding XDCL variable. In one compilation unit or group of units that will be combined for execution, a specific variable can have only one declaration that contains the XDCL attribute.

Declarations for a shared variable must match except for initialization. A variable declared with the XDCL attribute can be initialized and have different values assigned during program execution. A variable declared with the XREF attribute cannot be initialized but can be assigned values.

If you declare any attributes, the variable is assumed to be static in storage. If you don't declare any attributes, the variable is assumed to be automatic, unless you declare it at the outermost level of the module. (A variable declared at the outermost level is always static.)

Example:

Assume the following two modules have been compiled. When the loader loads the resulting object modules and satisfies externals, it allocates storage to FLAG, an XDCL variable, and initializes it to FALSE. When the loader finds the XREF variable FLAG in module TWO, it assigns the same storage. Thus, references to FLAG from either module refer to the same storage location.

```
MODULE one;
     :
  VAR
    flag: [XDCL] boolean := FALSE;
     :
MODEND one;
MODULE two;
     :
  VAR
    flag: [XREF] boolean;
     :
MODEND two;
```

## Storage

The storage attributes determine when storage is allocated and where storage is allocated.

*When Storage is Allocated*

There are two methods of allocating storage for variables: automatic and static. For an automatic variable, storage is allocated when the block containing the variable's declaration begins execution. Storage is released when execution of the block ends. If the block is entered again, storage is allocated again, and so on. When storage is released, the value of the variable is lost.

For a static variable, storage is allocated (and initialized, if that parameter is included) only once, at load time. Storage remains allocated throughout execution of the module. However, even though storage remains allocated, a static variable still follows normal scope rules. It can be accessed only within the block in which it is declared. A reference to a static variable from an outer block is an error even though storage for the static variable is still allocated.

The ability to declare a static variable is important, for example, in the case where an XDCL variable is referenced by a procedure before the procedure that declares the variable is executed. Because an XDCL variable is static (refer to Scope earlier in this chapter for further information), it is allocated space and is initialized immediately at load time; therefore, it is available to be referenced before execution of the procedure that actually declares it as XDCL.

A variable can be declared static with the STATIC attribute. It is assumed to be static if it is in the outermost level of a module or if it has any other attributes declared. In all other cases, CYBIL assumes the variable is automatic. Only a static variable can be initialized.

The period between the time storage for a variable is allocated and the time that storage is released is called the lifetime of the variable. It is defined in terms of modules and blocks. The lifetime of an automatic variable is the execution of the block in which it is declared. The lifetime of a static variable is the execution of the entire module. An attempt to reference a variable beyond its lifetime causes an error and unpredictable results.

The lifetime of a formal parameter in a procedure is the lifetime of the procedure in which it is a part. Storage space for the parameter is allocated when the procedure is called and released when the procedure finishes executing.

The lifetime of a pointer must be less than or equal to the lifetime of the data to which it is pointing.

The lifetime of a variable that is allocated using the storage management statements (described in chapter 5) is the time between the allocation of storage and the release of storage. A variable allocated by an automatic pointer (using the ALLOCATE statement) must be explicitly freed (using the FREE statement) before the block is left, or the space will not be released by the program. When the block is left, the pointer no longer exists and, therefore, the variable cannot be referenced. If the block is entered again, the previous pointer and the variable referenced by the pointer cannot be reclaimed. Therefore, it is recommended that you free such variables before leaving the block.

Example:

In this example, the variables COUNTER and FLAG will exist during execution of the entire module; however, they can be accessed only within program MAIN.

```
PROGRAM main;

  VAR
    counter: [STATIC] integer := 0,
    flag: [STATIC] boolean;
         :
PROCEND main;
```

*Where Storage is Allocated*

You can optionally specify that storage for a variable be allocated in a particular section. A section is a storage area that can hold variables sharing common access attributes, such as read-only variables or read/write variables. You can define the section and its access attributes yourself using the section declaration (discussed later in this chapter).

If you define a section with the section READ attribute, you define a read-only section in the hardware.[4] Any variable declared with that section's name as an attribute will reside in that read-only section. When you specify the name of a read-only section in a variable declaration, you must also include the variable access attribute READ.

In addition to any sections you define, CYBIL has several predefined sections. You cannot assign a variable to one of these sections explicitly, in the sense that you could include the section name as an attribute in your variable declarations. Instead, the variable is assigned to one of these predefined sections implicitly, based on its other attributes and characteristics. For example, all static variables that are not assigned to a user-defined section are automatically assigned to a section named $STATIC. The following are the CYBIL section names and their contents.

| **Section** | **Description** |
|---|---|
| $BINDING | The binding section that contains the links to external procedures and the data of the module. |
| CYB$DEFAULT_HEAP | The CYBIL default heap. |
| $LITERAL | Constants. |
| $PARAMETER | A subset of the $STACK section that contains parameter list variables. |
| $REGISTER | Variables that exist only in hardware registers. |
| $STACK | Automatic variables. |
| $STATIC | Static variables that are not already assigned to a user-defined section. |

4. A read-only section is a hardware feature. Data that resides in a physical area of the machine designated as a read-only section is protected by hardware, not by software. This feature is described in further detail in volume II of the virtual state hardware reference manual.

The SCL Object Code Management manual gives further information on sections regarding the object module format expected as input by the loader and the object library generator.

Example:

This example defines a read-only section named NUMBERS. The variable INPUT_NUMBER is a read-only variable that also resides in the section NUMBERS. In the variable declaration, the READ attribute causes the compiler to check that the variable is not written; the read-only section name, NUMBERS, causes the hardware to ensure that the variable is not written.

```
SECTION
  numbers: READ;

VAR
  input_number: [READ, numbers] integer := 100;
```

## Initialization

You can assign an initial value to a variable only if it is a static variable. The value can be a constant expression, an indefinite value constructor (described next), or a pointer to a global procedure. The value must be of the proper type and in the proper range. If you don't specify an initial value, the value of the variable is undefined.

An indefinite value constructor is essentially a list of values. It is used to assign values to the structured types: sets, arrays, and records. It allows you to specify several values rather than just one. Values listed in a value constructor are assigned in order (except for sets, which have no order). The types of the values must match the types of the components in the structure to which they are being assigned. An indefinite value constructor has the form

**[value** *{,value}...***]**

where value can be one of the following:

- A constant expression
- Another value constructor (that is, another list)

- The phrase

    REP number OF value

  which indicates the specified value is repeated the specified number of times

- The asterisk character (*), which indicates the element in the corresponding position is uninitialized

The REP phrase can be used only in arrays. The asterisk can be used only in arrays and records. For further information, refer to the descriptions of arrays and records in chapter 4.

If you assign an initial value to a string variable and the variable is longer than the initial value, spaces are added on the right of the initial value to fill the field. If the initial value is longer than the variable, the initial value is truncated on the right to fit the variable.

In a variant record, fields are initialized in order until a special variable called the tag field name is initialized. The tag field name is then used to determine the variant for the remaining field or fields in the record, and they are likewise initialized in order.

Depending on the attributes defined in the variable declaration, initialization is required, prohibited, or optional. Table 3-1 shows the initialization possible for various attributes.

**Table 3-1. Attributes and Initialization**

| Attributes Specified[1] | Initialization |
|---|---|
| None | Optional if static variable; prohibited if automatic variable. |
| READ | Required. |
| READ,STATIC | Required. |
| READ,XDCL | Required. |
| READ,STATIC,XDCL | Required. |
| READ,section_name | Required. |
| READ,XDCL,section_name | Required. |
| XREF | Prohibited. |
| XREF,READ | Prohibited. |
| XREF,STATIC | Prohibited. |
| XREF,READ,STATIC | Prohibited. |
| STATIC | Optional. |
| XDCL | Optional. |
| XDCL,STATIC | Optional. |
| section_name | Optional. |
| section_name,XDCL | Optional. |

1. The static attribute is assumed if any attributes are specified.

Example:

The variables declared in this example are inside program MAIN. Therefore, they are automatic unless they are declared with an attribute. TOTAL is automatic and as such cannot be initialized. COUNT is declared static and can be initialized. ALPHA and BETA are also static and can be initialized because they have other attributes declared.

```
PROGRAM main;
      :
  VAR
    total: integer,
    count: [STATIC] integer := 0,
    alpha,
    beta: [XDCL, READ] char := 'p';
      :
PROCEND main;
```

## Type Declaration

The standard data types that are defined in CYBIL are described in chapter 4. Any of these can be declared as a valid type within a variable declaration. The type declaration allows you to define a new data type and give it a name, or redefine an existing type with a new name. Then that name can be used as a valid type within a variable declaration.

Use this format for a type declaration:

**TYPE name = type** *{,name = type}...*;

**name**

Name to be given to the new type.

**type**

Any of the standard types defined by CYBIL or another user-defined type.

Once you define a type, you can use it to define yet another type. Thus, you can build a very complex type that can be referred to by a single name.

The type declaration is evaluated at compilation time. It does not occupy storage space during execution.

Examples:

In this example, INT is defined as a type consisting of all the integers; it is just a shortened name for a standard type. LETTERS is defined as a type consisting of the characters 'a' through 'z' only; this is a selective subset of the standard type characters. DEVICES is an ordinal type that in turn is used to define EQ_TABLE, a type consisting of an array of 10 elements. Any element in the type EQ_TABLE can have one of the ordinal values specified in DEVICES.

```
TYPE
  int = integer,
  letters = 'a' .. 'z',
  devices = (lp512, dk844, dk885, nt679),
  eq_table = array [1 .. 10] of devices;

VAR
  i: int,
  alpha: letters,
  table_1: eq_table,
  status_table: array [1 .. 3] of eq_table;
```

All of the variables in the preceding example could have been declared using variable declarations only, as in:

```
VAR
  i: integer,
  alpha: 'a' .. 'z',
  table_1: array [1 .. 10] of (lp512, dk844, dk885, nt679),
  status_table: array [1 .. 3] of array [1 .. 10] of
    (lp512, dk844, dk885, nt679);
```

However, it becomes cumbersome to declare a complex structure using only standard types. Defining your own types lets you avoid needless repetition and the increased possibility of errors. In addition, it makes code easier to maintain; to add a new device in the first example, you only need to add it in the type declaration, not in every variable declaration that contains devices.

## Section Declaration

A section is an optional working storage area that contains variables with common access attributes. You can define a section and its associated attributes with the section declaration. Including the section name in a variable declaration causes the variable to reside in that section.

Use this format for a section declaration:

**SECTION name** *{,name}...* **: attribute**
*{,name {,name}... : attribute}...*;

**name**

Name of the section.

**attribute**

The keyword READ or WRITE.

A section defined with the READ attribute is considered a read-only section.[5] A variable declared with that section's name will reside in read-only memory. In this case, the variable access attribute READ must also be included in the variable declaration. The section name causes hardware protection; the READ attribute causes compiler checking.

A section defined with the WRITE attribute contains variables that can be both read and written.

The initialization of variables declared with a section name depends on their attributes, as shown in table 3-1. Variables declared with a section name are static.

The names and contents of predefined CYBIL sections are given earlier in this section under Where Storage is Allocated. The SCL Object Code Management manual gives further information on sections regarding the object module format expected as input by the loader and the object library generator.

5. A read-only section is a hardware feature. Data that resides in a physical area of the machine designated as a read-only section is protected by hardware, not by software. This feature is described in further detail in volume II of the virtual state hardware reference manual.

Example:

Two sections are defined in this example: LETTERS is a read-only section and NUMBERS is a read/write section. The variable CONTROL_LETTER is a read-only variable that resides in LETTERS. The READ attribute is required because of the read-only section name. UPDATE_NUMBER is a variable that can be read or written, and resides in the section NUMBERS. In this example, it is also declared as an XDCL variable but this is not required.

```
SECTION
  letters: READ,
  numbers: WRITE;

VAR
  control_letter: [READ, letters] char := 'p',
  update_number: [XDCL, numbers] integer;
```

# Types 4

This chapter describes the standard types predefined by CYBIL.

# Types 4

There are many standard types defined within CYBIL. A variable can be assigned to (that is, be made an element of) any of these types. The type defines characteristics of the variable and what operations can be performed using the variable. In general, operations involving nonequivalent types are not allowed; one type cannot be used where another type is expected. Exceptions are noted in the descriptions of types that follow.

In this chapter, types are grouped into three major categories: basic, structured, and storage types.

Basic types are the most elementary. They can stand alone but are also used to build the more complex structures. The basic types are:

- Scalar types (integer, character, boolean, ordinal, and subrange)
- Floating-point types (real)
- Cell types
- Pointer types

Structured types are made from combinations of the basic types. The structured types are:

- Strings
- Arrays
- Records
- Sets

Storage types hold groups of components of various types. The storage types are:

- Heaps
- Sequences

Most types, when they are declared, have a fixed size. Strings, arrays, records, sequences, and heaps can also be declared with an adaptable size that is not fixed until execution. For this reason, they are sometimes called adaptable types. Adaptable strings, arrays, records, sequences, and heaps are discussed at the end of this chapter.

## Using Types

Types are used as parameters in two kinds of declarations: the variable declaration (to associate a type with a variable name) and the type declaration (to associate a type with a new type name). Both declarations are described in detail in chapter 3, but their basic formats are:

**VAR name :** *{ [attributes] }* **type** *{ := initial_value }* **;**

**TYPE name = type;**

The description of each type shown in this chapter includes the keyword and any additional information necessary to specify that type as a parameter. The keywords replace the generic word **type** in the variable and type declarations. For example, you would use the keyword INTEGER to specify an integer type. The variable declaration would be:

**VAR name :** *{ [attributes] }* **INTEGER** *{ := initial_value }* **;**

The type declaration would be:

**TYPE name = INTEGER;**

## Equivalent Types

As mentioned earlier in this chapter, operations involving nonequivalent types are not allowed. Two types can be equivalent, though, even if they don't appear to be identical. For example, two arrays can have different expressions defining their sizes, but the expressions may yield the same value. Rules for determining whether types are equivalent are given in the following descriptions of the types.

Adaptable types and bound variant record types (described under Records later in this chapter) actually define classes of related types that vary by a characteristic, such as size. Adaptable type variables, bound variant record type variables, and pointers to both types are fixed explicitly at execution time. These types are said to be potentially equivalent to any of the types to which they can adapt. That is, during compilation, references to adaptable types and bound variant record types are allowed wherever there is a reference to one of the types to which they can adapt. However, further type checking is done during execution when each type is fixed (assigned to a specific type). It is the current type of an adaptable or bound variant record type that determines what operations are valid for it at any given time.

## Basic Types

The following describes the basic types.

### Scalar Types

All scalar types have an order; that is, for every element of a scalar type you can find its predecessor and successor.

Scalar types are made up of five types:

- Integer
- Character
- Boolean
- Ordinal
- Subrange

## Integer

Use the keyword **INTEGER** to specify an integer type.

Integers range in value from $-(2^{63}-1)$ to $2^{63}-1$; that is, -7FFFFFFFFFFFFFFF hexadecimal through 7FFFFFFFFFFFFFFF hexadecimal. In general, the subrange type should be used rather than the integer type. This allows the compiler to perform more rigorous type checking and may reduce the amount of storage needed to hold the value.

The operations permitted on integers are assignment, addition, subtraction, multiplication, division (both quotient and remainder), all relational operations, and set membership. Refer to Operators in chapter 5 for further information on operations.

The functions $INTEGER and $REAL, described in chapter 6, convert between integer type and real type. The $CHAR function, also described in chapter 6, converts an integer value from 0 to 255 to a character according to its position in the ASCII collating sequence.

Example:

This example shows the definition of a new type named INT, which consists of elements of the type integer. The variable declaration declares variable I to be of type INT, which is the integer type just declared. Also declared as a variable is NUMBERS, which is explicitly of integer type. Because NUMBERS is static, it can be initialized.

```
TYPE
  int = integer;

VAR
  i: int,
  numbers: [STATIC] integer := 100;
```

## Character

Use the keyword **CHAR** to specify a character type.

An element of the character type can be any of the characters in the ASCII character set included in appendix C. It is always a single character; more than one character is considered a string. (A string is one of the structured types discussed later in this chapter. A string of length 1 can sometimes be used as a character. Refer to Substrings later in this chapter.)

The operations permitted on characters are assignment, all relational operations, and set membership. A character can be assigned and compared to a string of length 1. Refer to Operators in chapter 5 for further information on operations and to Strings later in this chapter for further information on string assignment.

The $INTEGER function described in chapter 6 converts a character value to an integer value based on its position in the ASCII collating sequence. The $CHAR function, also described in chapter 6, converts an integer value between 0 and 255 to a character in the ASCII collating sequence.

Example:

This example shows the definition of a new type named LETTERS, which consists of elements of the type character. The variable declaration declares variable ALPHA to be of type LETTERS, which is the character type; it is static and initialized to the character 'j'. The variable IDS is explicitly declared to be of type character.

```
TYPE
  letters = char;

VAR
  alpha: [STATIC] letters := 'j',
  ids: char;
```

## Boolean

Use the keyword **BOOLEAN** to specify a boolean type.

An element of the boolean type can have one of two values: FALSE or TRUE. As with other scalar types, boolean values are ordered. Their order is FALSE, TRUE. FALSE is always less than TRUE.

You get a boolean value by performing a relational operation on two objects of the same type. You can perform some, but not necessarily all, relational operations on every type except the following:

- Arrays or structures that contain an array as a component or field
- Variant records
- Sequences
- Heaps
- Records that contain a field of one of the preceding types

The operations permitted on boolean values are assignment, all relational operations, set membership, and boolean sum, product, difference, exclusive OR, and negation. Refer to Operators in chapter 5 for further information on operations.

The $INTEGER function described in chapter 6 converts a boolean value to an integer value. 0 is returned for FALSE; 1 is returned for TRUE.

Example:

This example shows the definition of a new type named STATUS, which consists of the boolean values FALSE and TRUE. The variable declaration declares variable CONTINUE to be of type STATUS; that is, it can be either FALSE or TRUE. The variable DEBUG is explicitly declared to be boolean and, because it is a read-only variable and therefore static, it can be initialized.

```
TYPE
  status = boolean;

VAR
  continue: status,
  debug: [READ] boolean := TRUE;
```

## Ordinal

The ordinal type differs from the other scalar types in that you define the elements within the type and their order. The term ordinal refers to the list of elements you define; the term ordinal name refers to an individual element within the ordinal.

Use this format to specify an ordinal:

**(name, name** *{,name...}* )

**name**

Name of an element within the ordinal. There must be at least two ordinal names. The maximum number of names in a single ordinal list is 16,384.

The order is given in ascending order from left to right.

Each ordinal name can be used in only one ordinal type. If you use a name in more than one ordinal, a compilation error occurs.

Ordinals are used to improve the readability and maintainability of programs. They allow you to use meaningful names within a program rather than, for example, map the names to a set of integers that are then used in the program to represent the names.

The operations permitted on ordinals are assignment, all relational operations, and set membership.

Two ordinal types are equivalent if they are defined in terms of the same ordinal type names.

The $INTEGER function described in chapter 6 converts an ordinal value (that is, a name) to an integer value based on its position within the defined ordinal. The first ordinal name has an integer value of 0, the second name an integer value of 1, and so on.

Examples:

In this example, the type declaration defines an ordinal type named COLORS, which consists of the elements RED, GREEN, and BLUE. The variable PRIMARY_COLORS is of COLORS type and therefore has the same elements. The variable WORK_DAYS explicitly declares the ordinal consisting of elements MONDAY through FRIDAY.

```
TYPE
  colors = (red, green, blue);

VAR
  primary_colors: colors,
  work_days: (monday, tuesday, wednesday, thursday,
    friday);
```

In the ordinal type COLORS, the following relationships hold:

RED < GREEN

RED < BLUE

GREEN < BLUE

You can find the predecessor and successor of every element of an ordinal. You can also map each element onto an integer using the $INTEGER function (described in chapter 6). For example, $INTEGER(RED) = 0; this is the first element of the ordinal.

The type declaration

```
TYPE
  primary_colors = (red, green, blue),
  hot_colors = (red, orange, yellow);
```

is in error because the name RED appears in two ordinal definitions.

## Subrange

A subrange is not a new type but a specified range of values within an existing scalar type. A variable defined by a subrange can take on only the values between and including the specified lower and upper bounds.

Use this format to specify a subrange:

**lowerbound . . upperbound**

**lowerbound**

Scalar expression specifying the lower bound of the subrange.

**upperbound**

Scalar expression specifying the upper bound of the subrange.

The lower bound must be less than or equal to the upper bound. Both bounds must be of the same scalar type.

The type of a subrange is the type of its lower and upper bounds. If a subrange completely encompasses its own type, it is said to be an improper subrange type. For example, the subrange

FALSE .. TRUE

is of type boolean and also contains every element of type boolean. It is equivalent to specifying the type itself. An improper subrange type is always equivalent to its own type.

Two subranges are equivalent if they have the same lower and upper bounds.

Subranges allow for additional error checking. Compilation options are available that cause the compiler to check assignments during program execution and issue an error if it finds a variable not within range. (Range checking is available as an option on the compiler call command and as a compiler directive. They are both described in chapter 8.) In addition, subranges improve readability. Because a subrange defines the valid range of values for a variable, it is more meaningful to you for documentation and maintenance.

The operations permitted on a subrange are the same as those permitted on its type (the type of its lower and upper bound).

Example:

This example shows the definition of a new type named LETTERS, which consists of the characters 'a' through 'z' only. It also defines an ordinal named COLORS, consisting of the colors listed. The variable declaration declares variable SCORES to consist of the numbers 0 through 100. The lower and upper bounds are of integer type, so the subrange is also an integer type. STATUS is a subrange of boolean values, which could have been declared simply as BOOLEAN. HOT_COLORS is a subrange of the ordinal type COLORS. It consists of the colors RED, ORANGE, and YELLOW.

```
TYPE
  letters = 'a' .. 'z',
  colors = (red, orange, yellow, white, green, blue);

VAR
  scores: 0 .. 100,
  status: FALSE .. TRUE,
  hot_colors: red .. yellow;
```

## Floating-Point Type

The floating-point type defines real numbers.

### Real

Use the keyword **REAL** to specify a real type.

Real numbers range in value from 4.8 * 10**(−1234) to 5.2 * 10**(1232).

The operations permitted on real types are assignment, addition, subtraction, multiplication, division, and all relational operations.

The functions $INTEGER and $REAL, described in chapter 6, convert between integer type and real type.

## Cell Type

The cell type represents the smallest storage location that is directly addressable by a pointer. On NOS/VE, a cell is an 8-bit byte within a 64-bit memory word.

Use the keyword **CELL** to specify a cell type.

Operations permitted on a cell type are assignment and comparison for equality and inequality.

## Pointer Types

A pointer represents the location of a value rather than the value itself. When you reference a pointer, you indirectly reference the object to which it is pointing.

Use this format to specify a pointer type:

**^ type**

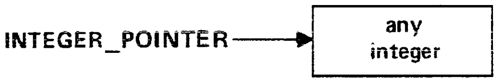
**type**

Type to which the pointer can point. It can be any defined type. With the exception of a pointer to cell type (discussed later in this chapter), the pointer can point only to objects of the type specified.

For example,

```
VAR
  integer_pointer: ^integer;
```

defines a pointer named INTEGER_POINTER that can point only to integers.



INTEG: 86/02/24

Use this format to specify the object of a pointer (that is, what the pointer points to):

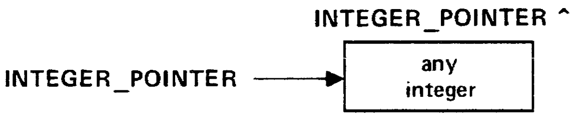
**pointer_name ^**

**pointer_name**
The name you gave the pointer in the variable declaration.

This preceding notation is called a pointer reference; it refers to the object to which pointer_name points. It can also be referred to as a dereference. For example,

```
integer_pointer^
```

identifies a location in memory; it is the location to which INTEGER_POINTER points.



POINT: 86/02/24

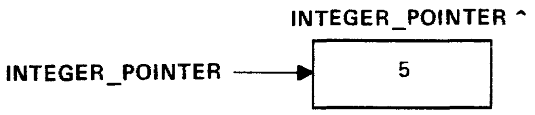
You can initialize or assign a value to the object of a pointer as you would any other variable; that is:

**pointer_name ^ := value;**

This assigns the specified value to the object that the pointer points to. For example,

```
integer_pointer^ := 5;
```

assigns the integer value 5 to the location that INTEGER_POINTER points to:



INTEG5: 86/02/24

You can assign the object of a pointer to a variable in the same way:

**variable := pointer_name ^;**

This takes the value of what pointer_name points to and assigns it to the variable. For example,

```
i := integer_pointer^;
```

assigns to I the contents of what INTEGER_POINTER points to, that is, 5.

If a pointer reference is to another pointer type variable, meaning that the pointer points to a pointer that in turn points to a variable, you can specify the variable in the format:

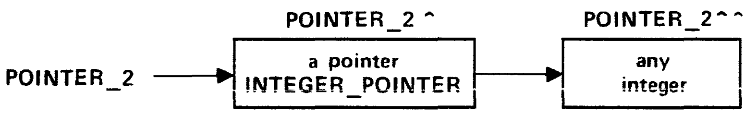
**pointer_name ^^**

For example, the declarations

```
TYPE
  integer_pointer = ^integer;

VAR
  pointer_2: ^integer_pointer;
```

can be pictured as follows:



POINT2: 86/02/24

POINTER_2 points to a pointer of type INTEGER_POINTER. INTEGER_POINTER points to integers. A reference to POINTER_2 ^ refers to the location of the pointer that in turn points to an integer. A reference to POINTER_2 ^^ refers to the location of the integer.

A pointer can be assigned any of the following values:

- The pointer constant NIL. NIL is the value of a pointer variable without an object; the variable is not currently assigned to any location. It can be assigned to or compared with any pointer of any type.
- The pointer symbol ^ followed by a variable of the type to which the pointer can point. If the variable is a formal value parameter, the pointer cannot be used to modify the variable.
- A pointer variable, which can be a component of a structured type as well as a valid parameter in a function.
- A function that returns a pointer as a value (such as the #LOC, #PTR, #REL, and #SEQ functions described in chapter 6).

Pointers allow you to manipulate storage dynamically. Using pointers, you can create and destroy variables while a program is executing. Memory is allocated when the variable is created and released when it is destroyed. Pointers also allow you to reference the variables without giving each a unique name.

Static pointers cannot point to value parameters or stack variables. [1] Stack pointers cannot point to value parameters or higher level stack variables. The lifetime of the pointer must be greater than or equal to the lifetime of the data. Parameter list pointers cannot point to value parameters or stack variables at the same or a higher level.

Permissible operations on pointers are assignment and comparison for equality and inequality.

Pointers to adaptable types (adaptable strings, arrays, records, sequences, and heaps) provide the only method for accessing objects of these types other than through formal parameters of a procedure. Specifically, pointers to adaptable types and pointers to bound variant records are used to access adaptable variables and bound variant records whose types have been fixed by an ALLOCATE, PUSH, or NEXT statement (described in chapter 5).

Pointers are equivalent if they are defined in terms of equivalent types. A pointer to a fixed type (as opposed to an adaptable type) can be assigned and compared to a pointer to an adaptable type or bound variant record if the adaptable type is potentially equivalent to the fixed type. (Refer to Equivalent Types earlier in this chapter for further information on potentially equivalent types.)

1. For further information on the run-time stack, refer to appendix F, The CYBIL Run-Time Environment.

Example:

The following example shows the declaration and manipulation of two pointer type variables. Comments appear to the right.

```
TYPE
  ptr = ^integer;             PTR can contain pointers to integers.

VAR
  i,
  j,
  k:integer,
  p1: ptr,                    P1 can contain pointers to integers.

  p2: ^p1,                    P2 can contain pointers to P1 (that is,
                              pointers that point to pointers to integers).
                              It could have been written as P2: ^^
                              INTEGER.

  b1,
  b2: boolean;

ALLOCATE p1;                  Allocates space for an integer (because that
                              is what P1 points to) and sets P1 to point
                              to that space.

ALLOCATE p2;                  Allocates space for a pointer that points to
                              an integer and sets P2 to point to that
                              pointer.

p1^ := 10;                    The space pointed to by P1 is set to 10.

p2^ := p1;                    The space pointed to by P2 is set to the
                              value of the pointer P1.

j := p1^;                     J is set to what P1 points to: the integer
                              10.

k := p2^^;                    K is set to the object of the pointer that
                              P2 points to. (Think of P2 ^^ as "P2 points
                              to a pointer; that pointer points to an
                              object." You are assigning that object to
                              K.) P2 points to P1, which points to the
                              integer 10.

b1 := j = k;                  J and K are both 10. B1 is TRUE.
```

| | |
|---|---|
| `b2 := p1^ = p2^^;` | P1 points to an integer. P2 points to the pointer (P1) that points to the same integer. Their values are the same and B2 is TRUE. |
| `p1 := NIL;` | P1 no longer points to anything. |
| `k := p1^;` | The statement is in error because P1 does not point to anything. |
| `IF p2 = NIL THEN`<br>`  k := k + 1;`<br>`IFEND;` | A valid statement. K is not incremented because P2 still points to P1. |
| `p1 := ^(i + j + 2 * k);` | An invalid statement. The location of an expression cannot be found. |

## Pointer to Cell

A pointer to cell type can take on values of any type.

Use this format to declare a pointer to a cell:

**^CELL**

A variable declared simply as a pointer type variable can take on as values only pointers to a single type, which is specified in the pointer's declaration. A variable declared as a pointer to cell variable has no such restrictions. It can take on values of any type. Also, any fixed or bound variant pointer variable can assume a value of pointer to cell.

Permissible operations on a pointer to a cell are assignment and comparison for equality and inequality. In addition, a pointer to a cell can be assigned to any pointer to a fixed or bound variant type. But the pointer to the fixed or bound variant type cannot have as its value a pointer to a variable that is not a cell type or, furthermore, whose type is not equivalent to the type to which the target of the assignment points. A pointer to a cell can be the target of assignment of any pointer to a fixed or bound variant type.

## Relative Pointer

Relative pointer types represent relative locations of components within an object with respect to the beginning of the object.

Use this format to specify a relative pointer:

**REL** *{ (parent_name) }* **^component_type**

*parent_name*

Name of the variable that contains the components being designated by relative pointers. Specify a string, array, record, heap, or sequence type (either fixed or adaptable). If it is omitted, the default heap is used.

**component_type**

Type of the component to which the relative pointer will point.

Relative pointers are generated using the standard function #REL (described in chapter 6). A relative pointer cannot be used to access data directly. Instead, the relative pointer must be converted to a direct pointer using the standard function #PTR (also described in chapter 6). The direct pointer can then be used to access the data.

Relative pointers have three major differences from the other pointers discussed in this chapter:

- Relative pointers may need less space than other pointers.
- A linked list or array of relative pointers (or some similar organization) within a parent type variable is still correct if the entire variable is assigned to another variable of the same parent type.
- Relative pointers are independent of the base address of the parent type variable.

Operations permitted on a relative pointer are assignment, comparison for equality and inequality, and the #PTR function. Relative pointers can be assigned and compared if they are of equivalent relative pointer types. Relative pointer types are equivalent if they are defined in terms of equivalent parent types and equivalent component types.

## Structured Types

Structured types are combinations of the basic types already described in this chapter (integer, character, boolean, ordinal, subrange, real, cell, and pointer). Even the structured types discussed here can be combined with each other but they are still essentially groups of the basic types. The structured types described in this section are:

- Strings
- Arrays
- Records
- Sets

### Strings

A string is one or more characters that can be identified and referenced as a whole by one name.

Use this format to specify a string type:

**STRING (length)**

**length**

A positive integer constant expression from 1 to 65,535.

If you specify an initial value in the variable declaration for a string, it can be:

- A string constant
- The name of a string constant declared with a constant declaration
- A constant expression (as described in chapter 2)

A string cannot be packed. [2] Two string types are equivalent if they have the same length.

2. Packing is a characteristic of arrays and records. When an array or record is declared as being packed, its components are mapped in storage to conserve storage space; otherwise, components are mapped to optimize access time.

The operations permitted on string types are assignment and comparison (all six relational operations). For further information, refer to Assigning and Comparing String Elements later in this chapter.

## Substrings

You can reference a part of a string (called a substring) or a single character of a string.

Use this format to reference a substring or single character:

**name (position** *{, length}* )

**name**

Name of the string.

**position**

Position within the string of the first character of the substring. (The position of the first character of the string is always 1.) Specify a positive integer expression less than or equal to the length of the string plus one; that is,

$1 \leq \text{position} \leq \text{string length} + 1$

If you specify string length plus one, the substring is an empty string.

*length*

Number of characters in the substring. Specify a nonnegative integer expression or * (the asterisk character). If you specify *, the substring consists of the character specified by the position parameter and all characters following it in the string. If you specify 0, the substring is an empty string. Omission causes 1 to be used.

A substring reference in the form

    name(position)

is a substring of length 1, a single character. In this form, it can be used anywhere a character expression is allowed. It can be:

- Compared with a character
- Tested for membership in a set of characters
- Used as the initial and/or final value in a FOR statement that is controlled by a character variable
- Used as a value in a CASE statement
- Used as an argument in the standard functions $INTEGER, SUCC, and PRED
- Assigned to a character variable
- Used as an actual parameter to a formal parameter of type character
- Used as an index value corresponding to a character type index in an array

A string constant, even if it is declared with a name in a constant (CONST) declaration, is not a variable. Therefore, substrings cannot be referenced in a string constant.

Examples:

If a string variable LETTERS is declared and initialized as follows

```
VAR
  letters: [STATIC] string (6) := 'abcdef';
```

the following substring references are valid:

| Substring | Comments |
|---|---|
| LETTERS(1) | Refers to 'a'. |
| LETTERS(6) | Refers to 'f'. |
| LETTERS(1,6) | Refers to the entire string. |
| LETTERS(1,*) | Refers to the entire string. |
| LETTERS(2,5) | Refers to 'bcdef'. |
| LETTERS(2,*) | Refers to 'bcdef'. |
| LETTERS(2,0) | Refers to an empty string ' '. |
| LETTERS(7,*) | Refers to an empty string ' '. |

LETTERS(0), LETTERS(8), and LETTERS(8,0) are illegal.

If a pointer variable is declared and initialized as follows

```
VAR
  string_ptr: [STATIC] ^string (6) := ^letters;
```

then STRING_PTR points to the string LETTERS and the pointer variable STRING_PTR^ can be used to make substring references similar to the variable LETTERS.

| Substring | Comments |
|---|---|
| STRING_PTR^(1) | Refers to 'a'. |
| STRING_PTR^(6) | Refers to 'f'. |
| STRING_PTR^(1,6) | Refers to the entire string. |
| STRING_PTR^(2,*) | Refers to 'bcdef'. |
| STRING_PTR^(2,0) | Refers to an empty string ' '. |

## Assigning and Comparing String Elements

You can assign or compare a character, substring, or string to a substring, string variable, or character variable. A character is treated as a string of length 1. You must specify the substring reference when assigning a character variable to a string.

If you assign a value that is longer than the substring or variable to which it is being assigned, the value is truncated on the right. If you assign a value that is shorter, spaces are added on the right to fill the field. This method is also used for comparing strings of different lengths.

If you assign a substring to a substring of the same variable, the fields cannot overlap or the results are undefined.

The concatenation operation CAT cannot be used with string variables.

Example:

Assume the string variable DAY is declared and initialized as follows:

```
VAR
  day: [STATIC] string (6) := 'monday';
```

The following assignments can be made:

```
short := day (1, 3);
empty := day (1, 0);
```

SHORT is assigned the string 'mon'. EMPTY is assigned a null string.

## Arrays

An array in CYBIL is a collection of data of the same type. You can access an array as a whole, using a single name, or you can access its elements individually.

Use this format to specify an array type:

*{PACKED}* **ARRAY [subscript_bounds] OF type**

*PACKED*

Optional packing parameter. When it is specified, the elements of the array are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If it is omitted, the array is unpacked; that is, the elements are mapped in storage to optimize access time rather than to conserve space. (The array itself is always mapped into an addressable memory location; that is, it starts on a word boundary or, in the case of a packed array in a record, on a byte boundary.) For further information on how data is stored in memory, refer to appendix E, Data Representation in Memory.

If the array contains structured types (such as records), the elements of that type (the fields in the records) are not automatically packed. The structured type itself must be declared packed.

**subscript_bounds**

Specifies the size of the array and what values you can use to refer to individual elements. Bounds can be any scalar type or subrange of a scalar type, and is often a subrange of integers.

**type**

Type of the elements within the array. The type can be any defined type, including another array, except an adaptable type (that is, an adaptable string, array, or record). All elements must be of the same type.

Elements of a packed array cannot be passed as reference (that is, VAR) parameters in programs, functions, or procedures.

Two array types are equivalent if they have the same packing attribute, equivalent subscript bounds, and equivalent component types.

The only operation permitted on an array type is assignment.

## Initializing Elements

An array can be initialized using an indefinite value constructor. An indefinite value constuctor is a list of values assigned in order to the elements of an array. The first value in the list is assigned to the first element, and so on. The number of values in the value constructor must be the same as the number of elements in the array. The type of the values must match the type of the elements in the array. An indefinite value constructor has the form

**[value** *{,value}...***]**

where value can be one of the following:

- A constant expression
- Another value constructor (that is, another list)
- The phrase

  REP number OF value

  which indicates the specified value is repeated the specified number of times
- The asterisk character (*), which indicates the element in the corresponding position is uninitialized

An indefinite value constructor can be used only for initialization; it cannot be used to assign values during program execution. Individual elements can be assigned during execution using the assignment statement (described in chapter 5).

## Referencing Elements

The array name alone refers to the entire structure.

Use this format to refer to an individual element of an array:

**array_name[subscript]**

**subscript**

A scalar expression within the range and of the type specified in the subscript_bounds field of the array declaration. This subscript specifies a particular element.

Examples:

This example shows the definition of a type named POS_TABLE, which is an array of 10 elements that can take on the values defined in POSITION. The variable declaration declares variable NUMBERS to be an array of five elements initialized to the values 1, 2, 3, 4, and 5 where 1 is the value of the first element, and so on. LETTERS is an array of 26 elements that can be any characters. BIG_TABLE is a 100-element array, of which each element is an array of 10 elements.

```
TYPE
  position = (boi, asis, eoi),
  pos_table = array [1 .. 10] of position;

VAR
  i: [STATIC] integer := 5,
  numbers: [STATIC] array [1 .. 5] of integer := [1, 2, 3, 4, 5],
  letters: array ['a' .. 'z'] of char,
  big_table: array [1 .. 100] of pos_table;
```

The declaration of BIG_TABLE is equivalent to:

```
VAR
  big_table: array [1 .. 100] of array [1 .. 10] of position;
```

You can reference individual elements using the following statements.

| | |
|---|---|
| `numbers [i]` | Refers to the fifth element of the array NUMBERS (similar to NUMBERS [5]). |
| `letters ['b'] := 'B';` | Sets the second element of the array LETTERS to the uppercase character B. |
| `big_table [13] [10] := asis;` | Sets the tenth element of the thirteenth array to ASIS. |

The following example shows the declaration and initialization of a two-dimensional array named DATA_TABLE. All the components of the third element of the array (which is an array itself) are set to 0. Notice that the third element of the last array, DATA_TABLE [4][3], is uninitialized.

```
TYPE
  innerarray = array [1 .. 5] of integer,
  twodim = array [1 .. 4] of innerarray;

VAR
  data_table: [STATIC] twodim := [[5, - 10, 2, 6, 3],
                                   [4, 11, 19, - 3, 6],
                                   [REP 5 of 0],
                                   [3, - 9, * , 4, 15]];
```

The following example demonstrates how a string can be passed to an array of characters:

```
VAR
   output_line: string (80),
   output_array: array [1..80] of char,
   i: integer;

FOR i := 1 to 80 DO
  output_array [i] := output_line (i);
FOREND;
```

## Records

Records are collections of data that can be of different types. You can access a record as a whole using a single name, or you can access elements individually.

A record has a fixed number of components, usually called fields, each with its own unique name. Different fields are used to indicate different data types or purposes.

There are two types of records: invariant records and variant records. Invariant records consist of fields that don't change in size or type. Variant records can contain fields that vary depending on the value of a key variable. Formats used for specifying both kinds of records are given later in this chapter.

Operations permitted on record types are assignment and, for invariant records only, comparison for equality and inequality. The invariant records being compared cannot contain arrays as fields.

## Invariant Records

An invariant record consists of fields that do not vary in size or type once they have been declared. They are called fixed or invariant fields.

Use this format to specify an invariant record:

> *{PACKED}* **RECORD**
>     **field_name :** *{ALIGNED {[offset MOD base]}}* **type**
>     *{,field_name : {ALIGNED {[offset MOD base]}} type}...*
> **RECEND**

*PACKED*

Optional packing parameter. When it is specified, the fields of a record are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If it is omitted, the record is unpacked; that is, the fields are mapped in storage to optimize access time rather than to conserve space. For further information on how data is stored in memory, refer to appendix E, Data Representation in Memory.

If one of the fields is a structured type (such as another record), the elements of that type are not packed automatically. The structured type itself must be declared packed.

**field_name**

Name identifying a particular field. The name must be unique within the record. Outside of the record declaration, it can be redefined.

*ALIGNED*

Optional alignment parameter. If specified, it can appear alone or with an offset, in the form:

*ALIGNED [offset MOD base]*

When a field is aligned, it is mapped in storage so that it is directly addressable. This means the field begins on an addressable boundary to facilitate rapid access to the field. This may negate some of the effect of packing the record. For further information, refer to Alignment later in this chapter.

*offset MOD base*

Optional offset to be used in conjunction with the ALIGNED parameter. This offset causes the field to be mapped to a particular hardware address relative to the specified base and offset. Specify a particular word or byte within a word. Base is evaluated first to find the word boundary; offset is then evaluated to determine the number of bytes offset within that word. Filler is created if necessary to ensure that the field begins on the specified word or byte.

*offset*

Byte offset within the word specified by base. Specify an integer constant less than base.

*base*

Word boundary. Specify an integer constant that is divisible by 8. For automatic variables, the base can only be 8.

**type**

Any defined type, including another record, other than an adaptable type.

Elements of a packed record cannot be passed as reference (that is, VAR) parameters in programs, functions, or procedures unless they are aligned.

The only operations possible on whole invariant records are assignment and comparison. A record can be assigned to another record if they are both of the same type. A record can also be compared to another record for equality or inequality if they are both of the same type. Invariant record types are the same if they have the same packing attributes, the same number of fields, and corresponding fields have the same field names, same alignment attribute, and equivalent types.

Example:

This example shows the definition of two new types, both records. The record named DATE has three fields that can hold, respectively, DAY, MONTH, and YEAR. The record named RECEIPTS appears to contain two fields, NAME and PAYMENT; but PAYMENT is itself a record consisting of the three fields in DATE, just described. Initialization of fields within records is discussed under Initializing Elements later in this chapter.

```
TYPE
  date = record
    day: 1 .. 31,
    month: string (4),
    year: 1900 .. 2100,
  recend,
  receipts = record
    name: string (40),
    payment: date,
  recend;
```

## Variant Records

A variant record contains fields that may vary in size, type, or number depending on the value of an optional tag field. These different fields are called variant fields or variants.

Use this format to specify a variant record:

*{PACKED} {BOUND}* **RECORD**
  *{fixed_field_name : {ALIGNED {[offset MOD base]}} type}...*[3]
  **CASE** *{tag_field_name : }* **tag_field_type OF**
  **= tag_field_value =**
    **variant_field**
  *{= tag_field_value =*
    *variant_field}...*
  **CASEND**
**RECEND**

*PACKED*

Optional packing parameter. When it is specified, the fields of a record are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If it is omitted, the record is unpacked; that is, the fields are mapped in storage to optimize access time rather than to conserve space. For further information on how data is stored in memory, refer to appendix E, Data Representation in Memory.

If a field is a structured type (such as another record), the elements of that type are not packed automatically. The structured type itself must be declared packed.

*BOUND*

Optional parameter indicating that this is a bound variant record. If specified, the tag_field_name parameter is required. Additional information on bound variant records follows the parameter descriptions.

3. When you specify more than one fixed field, you must separate them with commas.

*fixed_field_name*

Name of a fixed field (one that does not vary in size), as described under Invariant Records earlier in this chapter. The name must be unique within the record. Outside of the record declaration, it can be redefined. There can be zero or more fixed fields.

*ALIGNED*

Optional alignment parameter; the same as that for an invariant record. If specified, it can appear alone or with an offset in the form:

*ALIGNED [offset MOD base]*

When a field is aligned, it is mapped in storage so that it is directly addressable. This means the field begins on an addressable boundary to facilitate rapid access to the field. This may negate some of the effect of packing the record. For further information, refer to Alignment later in this chapter.

*offset MOD base*

Optional offset to be used in conjunction with the ALIGNED parameter, the same as that for an invariant record. This offset causes the field to be mapped to a particular hardware address relative to the specified base and offset. Specify a particular word or byte within a word. Base is evaluated first to find the word boundary; offset is then evaluated to determine the number of bytes offset within that word. Filler is created if necessary to ensure that the field begins on the specified word or byte.

*offset*

Byte offset within the word specified by base. Specify an integer constant less than base.

*base*

Word boundary. Specify an integer constant that is divisible by 8. For automatic variables, the base can only be 8.

*type*

Any defined type, including another record, other than an adaptable type.

*tag_field_name*

Optional parameter specifying the name of the variable that determines the variant. The current value of this variable determines which of the variant fields that follow will actually be used. If it is omitted, the variant that had the last assignment made to one of its fields is used. This parameter is required if the record is a bound variant record (BOUND is specified). Additional information is given following the parameter descriptions.

**tag_field_type**

Any scalar type. This type defines the values that the tag_field_value can have.

**tag_field_value**

Constant scalar expression or subrange. Specify one of the possible values that can be assigned to the variable specified by tag_field_name. It must be of the type and within the range specified by tag_field_type. Specifying a subrange has the same effect as listing each value separately.

**variant_field**

Zero or more fixed fields of the same form as that shown in the second line of this format. This field exists only if the current value of tag_field_name is the same as that in the tag_field_value associated with the variant_field. The last field can be a variant itself.

The variant fields must follow all invariant (fixed) fields in the record. The field following the reserved word CASE is called the tag_field_name. The tag_field_name can take on different values during execution. When its value matches one of the values specified in a tag_field_value, the variants associated with that tag_field_value are used. Variants themselves consist of zero or more fixed fields optionally followed by another variant. If the last field is itself a variant, it can have another CASE clause, tag_field_name, and so on.

The tag_field_name is an optional field. When it is omitted, no storage is assigned for the tag field. If the record has no tag field, you choose a variant by making an assignment to a subfield within a variant. The variant containing that subfield becomes the currently active variant. In a variant record without a tag field, all fields in a new active variant become undefined except the subfield that was just assigned. An attempt to access a variant field that is not currently active produces undefined results.

Space for a variant record is allocated using the largest possible variant.

Variant record types are equivalent if they have the same packing attribute, their fixed fields are equivalent (as defined for invariant record types), they have the same tag field names, their tag field types are equivalent, their tag field values are the same, and their corresponding variant fields are equivalent.

A bound variant record is specified by including the BOUND parameter; the tag_field_name is also required. A bound variant record type can be used only to define pointers for bound variant record types (that is, bound variant pointers). A variable of this type is always allocated in a sequence or heap, or in the run-time stack managed by the system.

When allocating a bound variant record, you must specify the tag field values that select the variation of the record. Only the specified space is allocated. The ALLOCATE statement in this case returns a bound variant pointer.

If a formal parameter of a procedure is a variant record type, the actual parameter cannot be a bound variant record type.

A record cannot be assigned to a variable that is a bound variant record type.

Bound variant record types are equivalent if they are defined in terms of equivalent, unbound records. A bound variant record type is never equivalent to a variant record type.

Example:

This example defines a type named SHAPE, which becomes the type of the tag field, in this case a variable named S. When S is equal to TRIANGLE, the record containing fields SIZE, INCLINATION, ANGLE1, and ANGLE2 is used as if it were the only record available. When the value of S changes, the record variant being used also changes.

```
TYPE
  shape = (triangle, rectangle, circle),
  angle = - 180 .. 180,
  figure = record
    x,
    y,
    area: real,
    case s: shape of
    = triangle =
      size: real,
      inclination,
      angle1,
      angle2: angle,
    = rectangle =
      side1,
      side2: integer,
      skew,
      angle3: angle,
    = circle =
      diameter: integer,
    casend,
  recend;
```

## Initializing Elements

A record can be initialized using an indefinite value constructor. An indefinite value constructor is a list of values assigned in order to the fields of a record. The first value in the list is assigned to the first field, or first element in a field, and so on. The type of the values must match the type of the elements in the field. An indefinite value constructor has the form

**[value** *{,value}...***]**

where value can be one of the following:

- A constant expression
- Another value constructor (that is, another list)
- The asterisk character (*), which indicates the element in the corresponding position is uninitialized

An indefinite value constructor can be used only for initialization; it cannot be used to assign values during program execution. Individual fields can be assigned during execution using the assignment statement (described in chapter 5).

Example:

The variable BIRTH_DAY, in this example, is a record with the fields described in the record type named DATE. It is initialized using an indefinite value constructor to the 24th day of August, 1950.

```
TYPE
  date = record
    day: 1 .. 31,
    month: string (4),
    year: 1900 .. 2100,
  recend;

VAR
  birth_day: [STATIC] date := [24, 'aug', 1950];
```

## Referencing Elements

The record name alone refers to the entire structure.

Use this format to access a field in a record:

**record_name.field_name** *{.sub_field_name}...*

**record_name**

Name of the record as declared in the variable declaration.

**field_name**

Name of the field to be accessed. If the field is an array, a reference to an individual element can also be included using the form:

**field_name[subscript]**

*sub_field_name*

Optional field name. Use this parameter if the field previously specified is itself a structured type, for example, another record. If the contained field is an array, you can include a reference to an individual element in the format:

*sub_field_name[subscript]*

Example:

The variable PROFILE is a record with the fields described in the record type STATS. In this example, PROFILE is initialized with the values in the indefinite value constructor in the variable declaration.

```
TYPE
  stats = record
    age: 6 .. 66,
    married: boolean,
    date: record
      day: 1 .. 31,
      month: 1 .. 12,
      year: 80 .. 90,
    recend,
  recend;

VAR
  profile: [STATIC] stats := [23, FALSE, [3, 5, 82]];
```

The following references can be made to fields:

| Field | Content |
|---|---|
| profile.age | 23 |
| profile.married | FALSE |
| profile.date.day | 3 |
| profile.date.month | 5 |
| profile.date.year | 82 |

## Alignment

Unpacked records and their fields are always aligned (that is, directly addressable). Even if it is packed, a record is always aligned (that is, the first field is directly addressable) unless it is an unaligned field within another packed structure. Fields in a packed record, however, are not aligned unless the ALIGNED attribute is explicitly included. Aligning the first field of a record aligns the entire record.

Unpacked records and their fields, because they are aligned, can always be passed as reference (that is, VAR) parameters in programs, functions, and procedures. Packed records must be aligned to be valid as reference parameters. Packed, unaligned records cannot be used.

## Sets

A set is a collection of elements that, unlike arrays and records, is always operated on as a single unit. Individual elements are never referenced.

Use this format to specify a set type:

**SET OF scalar_type**

**scalar_type**

Type of all the elements that will be within the set. Specify a scalar type or a subrange of a scalar type. The maximum number of elements in a set is 32,767.

All members of a set must be of the same type. Members within a set have no specific order; that is, order has no effect in any of the operations performed on sets.

Set types are equivalent if their elements have equivalent types.

Operations allowed on sets are assignment, intersection, union, difference, symmetric difference, negation, inclusion, identity, and membership. Refer to Operators in chapter 5 for further information on set operations. The SUCC and PRED functions are not defined for set types.

The difference (–) or symmetric difference (XOR) of two identical sets is the empty set. The empty set is contained in any set. For a given set, the complement of the empty set, –[ ], is the full set.

## Initializing and Assigning Elements

Values can be assigned to a set using an indefinite value constructor or a set value constructor. An indefinite value constructor can be used only for initialization; a set value constructor can be used for both initialization and assignment during program execution.

An indefinite value constructor is a list of values assigned to the set. The type of the values must match the type of the set.

Use this format to specify an indefinite value constructor:

**[value** *{,value}...***]**

**value**

Constant expression or another indefinite value constructor (that is, another list).

A set value constructor constructs a set through explicit assignment. Use this format to specify a set value constructor:

**$name [** *{ value {,value}...}* **]**

**name**

Name of the set type. The dollar sign ($) must precede the name to indicate a set value constructor.

*value*

Expression of the same type as that specified for the set. When used in initialization, only constants or constant expressions are valid. The empty set can be specified by [ ].

A set value constructor can be used wherever an expression can be used.

Example:

This example shows the declaration of a variable named ODD, which is a type of a set of integers from 0 to 10. It is initialized with an indefinite value constructor assigning the integers 1, 3, and 5 to the set. The variable VOWELS is a set that can contain any of the letters 'a' through 'z'. It is assigned the letters 'a', 'e', 'i', 'o' and 'u' using a set value constructor. It constructs a set of type C, which contains the specified letters; then that set is assigned to the set VOWELS. The variables LIST_1 and LIST_2 are sets that can contain any characters. LIST_1 is assigned, using a set value constructor, the letters 'x', 'y', and 'z'. LIST_2 is assigned the complement of 'x', 'y', and 'z', that is, a set consisting of every character except the letters 'x', 'y', and 'z'.

```
TYPE
  a = set of 0 .. 10,
  c = set of 'a' .. 'z',
  ch = set of char;

VAR
  odd: [STATIC] a := [1, 3, 5],
  vowels: c,
  list_1,
  list_2: ch;
    :
vowels := $c ['a', 'e', 'i', 'o', 'u'];
list_1 := $ch ['x', 'y', 'z'];
list_2 := - $ch ['x', 'y', 'z'];
```

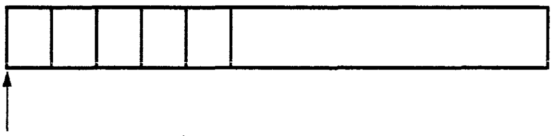
## Storage Types

Storage types represent structures to which variables can be added, deleted, and referenced under program control. (The statements used to access the storage types are described under Storage Management Statements in chapter 5.) There are two storage types:

- Sequences
- Heaps

### Sequences

A sequence type is a storage structure whose components are referenced sequentially using pointers. It can be pictured as follows:



SEQNCE: 86/08/08

These pointers are constructed using the RESET and NEXT statements (described in chapter 5). The RESET statement moves the pointer to the beginning of the sequence or to a specific variable within the sequence. The NEXT statement moves the pointer to the next available space.

Use this format to specify a sequence type:

**SEQ** *({REP number OF}* **type** *{,{REP number OF} type}...)*

*number*

Positive integer constant expression. This is an optional parameter specifying the number of repetitions of the specified type.

**type**

Fixed type that can be a user-defined type name; one of the predefined types integer, character, boolean, real, or cell; or a structured type using the preceding types.

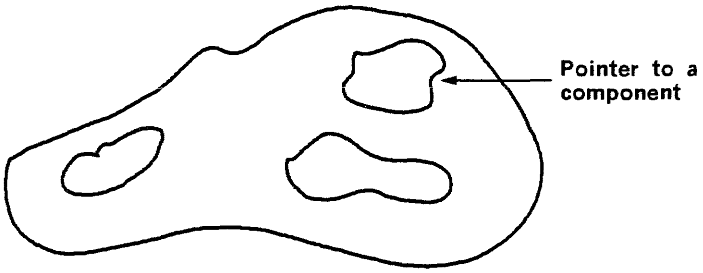
You can repeat the phrase *REP number OF type* as many times as desired. It specifies that storage must be available to hold the indicated number of occurrences of the named types simultaneously. The types that are actually stored in a sequence do not have to be the same as the types specified in the declaration, but adequate space must have been allocated to hold those types in the declaration. In other words, if a sequence is declared with several repetitions of integer type, the space to hold these integers has to be available, but it might actually hold strings or boolean values.

Sequence types are equivalent if they have the same number of REP phrases and corresponding phrases are equivalent. Two REP phrases are equivalent if they have the same number of repetitions of equivalent types.

Assignment to another sequence is the only operation permitted on sequences.

## Heaps

A heap type is a storage structure whose components are referenced using pointers but, unlike a sequence, they are not allocated and referenced sequentially. A heap can be pictured as follows:



HEAP: 86/08/08

The components of a heap are allocated explicitly using the ALLOCATE statement, which also constructs pointers that you can use to reference the components. The components of a heap are released using the FREE and RESET statements (described in chapter 5).

Use this format to specify a heap type:

**HEAP** *({REP number OF}* **type** *{,{REP number OF} type}...)*

*number*

Positive integer constant expression. This is an optional parameter specifying the number of repetitions of the specified type.

**type**

Fixed type that can be a user-defined type name; one of the predefined types integer, character, boolean, real, or cell; or a structured type using the preceding types.

You can repeat the phrase *REP number OF type* as many times as desired. It specifies that storage must be available to hold the indicated number of occurrences of the named types simultaneously. The types that are actually stored in a heap do not have to be the same as the types specified in the declaration, but adequate space must have been allocated to hold those types in the declaration. In other words, if a heap is declared with several repetitions of integer type, the space to hold these integers has to be available, but it might actually hold strings or boolean values.

Heap types are equivalent if they have the same number of REP phrases and corresponding phrases are equivalent. Two REP phrases are equivalent if they have the same number of repetitions of equivalent types.

The default heap can be managed with the ALLOCATE and FREE statements in the same way as a user-defined heap. For further information, refer to the descriptions of these statements in chapter 5.

## Adaptable Types

An adaptable type has indefinite size or bounds; it adapts to data of the same type but of different sizes and bounds. The types described thus far in this chapter are fixed types. An adaptable type differs from a fixed type in that the storage required for a fixed type is constant and can be determined before execution. Storage for an adaptable type is determined during program execution.

An adaptable type can be a string, array, record, sequence, or heap and can define formal parameters in a procedure and adaptable pointers. Pointers are the mechanism used for referencing adaptable variables.

The size of an adaptable type must be set during execution. This can be done in one of three ways:

- If the adaptable type is a formal parameter to a procedure or function, the size is set by the actual parameters when the procedure or function is called. You can determine the length of an actual parameter string using the STRLENGTH function, and the bounds of an actual parameter array using the UPPERBOUND and LOWERBOUND functions. (For further information, refer to the description of the appropriate function in chapter 6.)

- If the adaptable pointer type is on the left side of an assignment statement, the size is set by the assignment operation. It can be assigned any pointer whose current type is one of the types that the adaptable type can take on.

- An adaptable type can also be set explicitly using the storage management statements (described in chapter 5).

An adaptable type is declared with an asterisk taking the place of the size or bounds normally found in the type or variable declaration.

## Adaptable Strings

Use this format to specify an adaptable string:

> **STRING** ( * *{<= length}* )

*length*

Optional parameter specifying the maximum length of the adaptable string. If it is omitted, 65,535 characters is assumed.

If the string exceeds the maximum allowable length, an error occurs.

Two adaptable string types are always equivalent.

## Adaptable Arrays

Use this format to specify an adaptable array:

> *{PACKED}* **ARRAY** [*{lower_bound ..}* *] **OF type**

*PACKED*

Optional packing parameter. When it is specified, the elements of the array are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If it is omitted, the array is unpacked; that is, the elements are mapped in storage to optimize access time rather than to conserve space. (The array itself is always mapped into an addressable memory location.) For further information on how data is stored in memory, refer to appendix E, Data Representation in Memory.

If the array contains structured types (such as records), the elements of that type (the fields in the records) are not automatically packed. The structured type itself must be declared packed.

*lower_bound*

Constant integer expression that specifies the lower bound of the adaptable array. This parameter is optional, but its use is encouraged. Omission of this parameter (only the * appears) indicates it is an adaptable bound of type integer.

**type**

Type of the elements within the array. The type can be any defined type other then adaptable (that is, an adaptable string, array, record, sequence, or heap). All elements must be of the same type.

Only one dimension can be adaptable in an array and that dimension must be outermost (first one in the declaration).

Adaptable arrays adapt to a specific range of subscripts. An adaptable array can adapt to any array with the same packing attribute, equivalent subscript bounds, and equivalent component types. If a lower bound is specified in the adaptable array declaration, both arrays must also have the same lower bound.

Adaptable array types are equivalent if they have the same packing attributes and equivalent component types, and if their corresponding array and component subscript bounds are equivalent. Two subscript bounds that contain asterisks only are always equivalent. Two subscript bounds that contain identical lower bounds are equivalent.

## Adaptable Records

An adaptable record contains zero or more fixed fields followed by one adaptable field that is of an adaptable type.

Use this format to specify an adaptable record:

```
{PACKED} RECORD
   {fixed_field_name : {ALIGNED {[offset MOD base]}} type}...⁴
   adaptable_field_name : {ALIGNED {[offset MOD base]}}
      adaptable_type
RECEND
```

*PACKED*

Optional packing parameter. When it is specified, the fields of a record are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If it is omitted, the record is unpacked; that is, the fields are mapped in storage to optimize access time rather than to conserve space. For further information on how data is stored in memory, refer to appendix E, Data Representation in Memory.

If a field is a structured type (such as another record), the elements of that type are not packed automatically. The structured type itself must be declared packed.

*fixed_field_name*

Name identifying a particular fixed field. The name must be unique within the record.

*ALIGNED*

Optional alignment parameter. If it is specified, it can appear alone, or with an offset in the form:

*ALIGNED [offset MOD base]*

When a field is aligned, it is mapped in storage so that it is directly addressable. This means the field begins on an addressable boundary to facilitate rapid access to the field. This may negate some of the effect of packing the record. For further information, refer to Alignment earlier in this chapter.

4. If you specify more than one fixed (nonadaptable) field, you must separate them with commas.

*[offset MOD base]*

Optional offset to be used in conjunction with the ALIGNED parameter. This offset causes the field to be mapped to a particular hardware address relative to the specified base and offset. Filler is created if necessary to ensure that the field begins on the specified addressable unit.

*offset*

An integer constant. Offset must be less than base.

*base*

An integer constant that must be divisible by 8. For automatic variables, the base can only be 8.

*type*

Any defined type, including another record, other than an adaptable type.

**adaptable_field_name**

Name identifying the adaptable field.

**adaptable_type**

An adaptable type.

An adaptable record can adapt to any record whose types are the same except for the last field. That last field must be one to which the adaptable field can adapt.

Two adaptable record types are equivalent if they have the same packing attributes, the same alignment, the same number of fields, and corresponding fields with identical names and equivalent types.

## Adaptable Sequences

Use this format to specify an adaptable sequence:

**SEQ (*)**

An adaptable sequence can adapt to a sequence of any size. Two adaptable sequence types are always equivalent.

## Adaptable Heaps

Use this format to specify an adaptable heap:

**HEAP (*)**

An adaptable heap can adapt to a heap of any size. Two adaptable heap types are always equivalent.

# Expressions and Statements 5

This chapter describes expressions and statements that can be used within a CYBIL program, procedure, or function.

# Expressions and Statements 5

## Expressions

Expressions are made up of operands and operators. Operators act on operands to produce new values. (Constant expressions are evaluated to provide values for constants. Refer also to Constant Expressions in chapter 2.)

In general, operations involving nonequivalent types are not allowed; one type cannot be used where another type is expected. Exceptions are noted in the following descriptions.

### Operands

Operands hold or represent the values to be used during evaluation of an expression. An operand can be a variable, constant, name of a constant, set value constructor, function reference (either standard function or user-defined function), pointer to a procedure name, pointer to a variable, or another expression enclosed in parentheses.

The value of a variable being used as an operand is the last value assigned to it. A constant name is replaced by the constant value associated with it in the constant declaration.

A function reference causes the function to be executed; the value returned by the function takes the place of the function reference in the expression.

### Operators

Operators cause an action to be performed on one operand or a pair of operands. Many of the operators can be used only on basic types; they will be noted in their individual descriptions. Some operators can be used on sets. Although they are discussed in the individual descriptions that follow, for a more detailed description also refer to Set Operators later in this chapter.

An operation on a variable or component of a variable that has an undefined value will produce an undefined result.

There are five kinds of operators, many of which are identified by reserved symbols. They are listed next in the order in which they are evaluated, from highest to lowest precedence.

- Negation operator (NOT)
- Multiplication operators ( * , DIV, / , MOD, and AND)
- Sign operators ( + and −)
- Addition operators ( + , − , OR, and XOR)
- Relational operators ( < , <= , > , >= , = , <> , and IN)

In relational operators that consist of two symbols (that is, <=, >=, and <>), do not separate the symbols with a space or any other character; the symbols must appear together.

When an expression contains two or more operators of the same precedence, operations are performed from left to right. The only way to explicitly change the order of evaluation is to use parentheses. Parentheses specify that the expression inside them should be evaluated first.

## Negation Operator

The negation operator, NOT, applies only to boolean operands.

NOT TRUE equals FALSE. NOT FALSE equals TRUE.

## Multiplication Operators

The multiplication operators perform multiplication and set intersection (*), integer quotient division (DIV), real quotient division (/), remainder division (MOD), and the logical AND operation (AND). Table 5-1 shows the multiplication operators, the permissible types of their operands, and the type of result they produce.

**Table 5-1. Multiplication Operators**

| Operator | Operation | Type of Operand | Type of Result |
|---|---|---|---|
| * | Multiplication | Integer or subrange of integer | Integer |
| | | Real | Real |
| * | Set intersection | Set of a scalar type | Set of the same type |
| DIV | Integer quotient[1] | Integer or subrange of integer | Integer |
| / | Real quotient | Real | Real |
| MOD | Remainder[2] | Integer or subrange of integer | Integer |
| AND | Logical AND[3] | Boolean | Boolean |

1. Integer quotient refers to the whole number that results from a division operation; the remainder is ignored. A more formal definition is: for positive integers a, b, and n, a DIV b = n where n is the largest integer so that b * n <= a.

For one or two negative integers,

(-a) DIV b = (a) DIV (-b) = - (a DIV b) and
(-a) DIV (-b) = a DIV b

2. Remainder refers to the remainder of a division operation. A more formal definition is:

a MOD b = a - (a DIV b) * b

3. The logical AND operation is evaluated as follows:

TRUE AND FALSE = FALSE
TRUE AND TRUE = TRUE
FALSE AND FALSE = FALSE
FALSE AND TRUE = FALSE

When the first operand is FALSE, the second operand is never evaluated.

## Sign Operators

The sign operators perform the identity operation (+) and sign inversion and set complement operation (−). Table 5-2 shows the sign operators, the permissible types of their operands, and the type of result they produce.

Table 5-2. Sign Operators

| Operator | Operation | Type of Operand | Type of Result |
|---|---|---|---|
| + | Identity (indicates a positive operand) | Integer | Integer |
| | | Real | Real |
| - | Sign inversion (indicates a negative operand) | Integer | Integer |
| | | Real | Real |
| - | Set complement | Set of a scalar type | Set of the same type |

## Addition Operators

The addition operators perform addition and set union (+), subtraction, boolean difference, and set difference (–), the logical OR operation (OR), and the exclusive OR operation (XOR). Table 5-3 shows the addition operators, the permissible types of their operands, and the type of result they produce.

**Table 5-3. Addition Operators**

| Operator | Operation | Type of Operand | Type of Result |
|---|---|---|---|
| + | Addition | Integer or subrange of integer | Integer |
| | | Real | Real |
| + | Set union | Set of a scalar type | Set of the same type |
| – | Subtraction | Integer or subrange of integer | Integer |
| | | Real | Real |
| – | Boolean difference[1] | Boolean | Boolean |
| – | Set difference | Set of a scalar type | Set of the same type |

1. The boolean difference operation is evaluated as follows:

```
TRUE - TRUE = FALSE
TRUE - FALSE = TRUE
FALSE - TRUE = FALSE
FALSE - FALSE = FALSE
```

*(Continued)*

**Table 5-3. Addition Operators** *(Continued)*

| **Operator** | **Operation** | **Type of Operand** | **Type of Result** |
|---|---|---|---|
| OR | Logical OR[2] | Boolean | Boolean |
| XOR | Exclusive OR[3] | Boolean | Boolean |
| XOR | Symmetric difference | Set of a scalar type | Set of the same type |

2. The logical OR operation is evaluated as follows:

TRUE OR TRUE = TRUE
TRUE OR FALSE = TRUE
FALSE OR TRUE = TRUE
FALSE OR FALSE = FALSE

When the first operand is TRUE, the second operand is never evaluated.

3. The exclusive OR operation is evaluated as follows:

TRUE XOR TRUE = FALSE
TRUE XOR FALSE = TRUE
FALSE XOR TRUE = TRUE
FALSE XOR FALSE = FALSE

## Relational Operators

The relational operators (<, <=, >, >=, =, <>, and IN) test whether the following given conditions are true or false: less than (<), less than or equal to or subset of a set (<=), greater than (>), greater than or equal to or a superset of a set (>=), equal to or set identity (=), not equal to or set inequality (<>), and set membership (IN).

Because relational operators are valid on so many different types, some special points about each type are noted next. Following these comments, table 5-4 lists the relational operators and the permissible types of their operands; they always produce a boolean type result.

### *Comparison of Scalar Types*

The comparison operators ( < , <= , > , >= , = , and <> ) are allowed only between operands of the same scalar type or between a substring of length 1 and a character.

For integer type operands, the relationships all have their usual meaning.

For character type operands, each character is essentially mapped to its corresponding integer value according to the ASCII collating sequence. (This is the same operation performed by the $INTEGER function described in chapter 6.) The operands and relational operators are then evaluated using the characters' integer values.

For boolean type operands, FALSE is always considered to be less than TRUE.

For ordinal type operands, operands are equal only if they are the same value; otherwise, they are not equal. For the other relational operators, each ordinal is essentially mapped to the corresponding integer value of its position in the ordinal list where it is defined. (This is the same operation performed by the $INTEGER function described in chapter 6.) The operands and relational operators are then evaluated using the ordinals' integer values. For an example, refer to the discussion of ordinal types under Scalar Types in chapter 4.

Operands that are a subrange of a scalar type can be compared with operands of the same type, including another subrange of the same type.

### *Comparison of Floating-Point Types*

All of the comparison operators are valid between operands of the real type.

### *Comparison of Pointer Types*

Two pointers can be compared if they are pointers to equivalent or potentially equivalent types. (For further information on equivalent types, refer to Equivalent Types in chapter 4.) For potentially equivalent types, one or both of the pointers can be pointers to adaptable or bound variant types. The current type of such a pointer must be equivalent to the type of the pointer with which it is being compared; if it is not, the operation is undefined.

Pointers can be compared for equality and inequality only. Two pointers are equal if they designate the same variable or if they both have the value NIL. A pointer of any type can be compared with the value NIL. Two pointers to a procedure are equal if they designate the same declaration of a procedure.

### *Comparison of Relative Pointers*

Two relative pointers can be compared only if they are of equivalent types. Two relative pointers are equal if they can be converted to equal pointers using the #PTR function (described in chapter 6).

### *Comparison of String Types*

All of the comparison operators are valid between operands that are strings. If the lengths of the two string operands are unequal, spaces are added to the right of the shorter string to fill the field.

Strings are compared character by character from left to right; that is, each character from one string is compared with the character in the corresponding position of the second string. Each character is compared using the same method as for operands of character type; the integer value of the character, when mapped to the ASCII collating sequence, is used.

### *Comparison of Sets and Set Membership*

Comparison operators have slightly different meanings for sets than for other types. The only comparison operators valid for sets are: = (identical to), <> (different from), <= (the left operand is contained in the right operand), and >= (the left operand contains the right operand). These operators are valid between two sets of the same type. Their exact meanings are detailed later in this chapter under Set Operators.

The other relational operator for sets is IN. A specified operand is IN a set if that operand is a member of the set. The set must be of the same type or a subrange of the same type as the operand. The operand can be a subrange of the type of the set.

### *Comparison of Other Types*

Invariant records can be compared for equality and inequality only. Two equivalent records are equal if their corresponding fields are equal.

The following types cannot be compared:

- Arrays or structures that contain an array as a component or field
- Variant records
- Sequences
- Heaps
- Records that contain a field of one of the preceding types

However, pointers to these types can be compared.

**Table 5-4. Relational Operators**

| Operator | Operation | Type of Left Operand | Type of Right Operand |
|---|---|---|---|
| < | Less than | Any scalar type | The same scalar type |
| | | Real | Real |
| <= | Less than or equal to | A string | A string of the same length |
| > | Greater than | A string of length 1[1] | A character |
| >= | Greater than or equal to | | |
| = | Equal to | A character | A string of length 1[1] |
| <> | Not equal to | | |
| IN | Set membership | Any scalar type | A set of the same type |
| | | Real | A set of real type |
| | | A string of length 1[1] | A set of character type |

1. The string of length 1 has the form

STRING(position)

where the length is implied. The form

STRING(position,1)

is not valid in this case.

*(Continued)*

**Table 5-4. Relational Operators** *(Continued)*

| Operator | Operation | Type of Left Operand | Type of Right Operand |
|---|---|---|---|
| = | Equality (also called identity) | A set of any scalar type | A set of the same type |
| <> | Inequality | A set of real type | A set of real type |
| <= | Is contained in | | |
| >= | Contains | | |
| =<br><> | Equality<br>Inequality | A nonvariant record type containing no arrays | The same type |
| | | Any pointer type or the value NIL | The same type or the value NIL |

## Set Operators

The set operators have already been mentioned briefly in the preceding sections on multiplication, sign, addition, and relational operators. This section discusses all of them and explains how they are used with sets.

The set operators perform assignment, union (+), intersection (*), difference (–), symmetric difference (XOR), negation (–), identity or equality (=), inequality (<>), inclusion (<=), containment (>=), and membership (IN).

Assignment is discussed under Sets in chapter 4. The next five operations (union, intersection, difference, symmetric difference, and negation) all produce results that are sets (they are described in table 5-5). The remaining operations (identity, inequality, inclusion, containment, and membership) produce boolean results (they are described in table 5-6).

The relational operations described in table 5-6 occur only after any operations described in table 5-5 have been performed.

**Table 5-5. Operations That Produce Sets**

| Operator | Operation | Resulting Set |
|---|---|---|
| + | Union | All members of both sets. The result of A + B is all elements of sets A and B. |
| - | Difference | Members in the lefthand set that are not in the righthand set. The result of A - B is the elements of A that are not in B. This operation differs from negation in that two operands are present. |
| * | Intersection | Members that are in both sets. The result of A * B is all elements that are in both A and B. |
| - | Negation (complement) | Members of the set's type that are not in the set. The result of -A is all elements of A's type that are not in A. This operation differs from the difference operation in that only one operand is present. |
| XOR | Symmetric difference | Members of either but not both sets. The result of A XOR B is all elements in A or B that are not common to both A and B. |

**Table 5-6. Operations That Produce Boolean Results**

| Operator | Operation | Resulting Value |
|---|---|---|
| = | Equality (identity) | TRUE if every member of one set is present in the other set and vice versa. A = B is TRUE if every element of A is in B and every element of B is in A. It is also TRUE if A and B are both empty sets. In any other case, it is FALSE. |
| <> | Inequality | TRUE if not every member of one set is a member of the other set. A <> B is TRUE if A = B is FALSE. |
| <= | Inclusion | TRUE if every member of the lefthand set is also a member of the righthand set. A <= B is TRUE if every element of A is in B. It is also TRUE if A is an empty set. In all other cases, it is FALSE. |
| >= | Containment | TRUE if every member of the righthand set is also a member of the lefthand set. A >= B is TRUE if every element of B is in A (that is, B <= A). |
| IN | Membership | TRUE if the scalar is of the same type as the type of the set, and is an element within the set. This operation differs somewhat from the others in that it can specify a value or a variable as an operand, rather than a set. It has the form<br><br>scalar IN set<br><br>where scalar can be a value (including a subrange) or a variable. A IN B is TRUE if A is the same type as the set B and A is an element of B. |

## Statements

Statements specify actions to be performed. Unlike declarations, statements can be executed. They can appear only in a program, procedure, or function.

A statement list is an ordered sequence of statements. In a statement list, a statement is separated from the one following it by a semicolon. Two consecutive semicolons indicate an empty statement, which means no action.

Statements can be divided into four types depending on their purpose or nature:

- Assignment
- Structured
- Control
- Storage management

### Assignment Statement

The assignment statement assigns a value to a variable.

Use this format for the assignment statement:

**name := expression;**

**name**

Name of a variable previously declared.

**expression**

An expression that meets the requirements stated earlier in this chapter. Any constant or variable contained in the expression must be defined and have a value assigned.

This statement is similar to the initialization part of the VAR declaration where you can assign an initial value to a variable. (For further information on initialization, refer to Variable Declaration in chapter 3.) The assignment statement allows you to change that value at any point in the program. The expression is evaluated and the result becomes the current value of the named variable.

The variable cannot be:

- A read-only variable
- A formal value parameter of the procedure that contains the assignment statement
- A bound variant record
- The tag field name of a bound variant record
- A heap
- An array or record that contains a heap

The type of the expression must be equivalent to the type of the variable, with the exceptions discussed next. Both types can be subranges of equivalent types.

A character, string, or substring variable can be assigned the value of a character expression, a string, or a substring. If you assign a value that is shorter than the variable or substring to which it is being assigned, spaces are added to the right of the shorter string to fill the field. If you assign a value that is longer than the variable or substring, the value is truncated on the right. Assigning strings or substrings that overlap is not a valid operation, for example, STRING_1 := STRING_1(3,7); results are unpredictable.

If the variable is a pointer, its scope must be less than or equal to the scope of the data to which it is pointing. For example, a static pointer variable should not point to an automatic variable local to a procedure. When the procedure is left, the pointer variable will be pointing at undefined data.

A pointer to a bound variant record can be assigned a pointer to a variant record that is not bound and is otherwise equivalent.

An adaptable pointer can be assigned either a pointer to a type to which it can adapt, or an adaptable pointer that has been adapted to one of those types. Both the type of the expression and its value are assigned, thus setting the current type of the adaptable pointer.

Any fixed pointer except a pointer to sequence can be assigned a pointer to cell. After the assignment, the #LOC function (described in chapter 6) performed on the fixed pointer would return the same value as the pointer to cell.

A pointer to cell can be assigned any pointer type. The value assigned is a pointer to the first cell allocated for the variable to which the pointer being assigned points.

When assigning pointers, remember that the object of a pointer has a different lifetime than the pointer variable. Automatic variables are released when the block in which they are declared has been executed. Allocated variables no longer exist when they are explicitly released with the FREE statement. An attempt to reference a variable beyond its lifetime causes an error and unpredictable results to occur.

A variant record can be assigned a bound variant record of types that are otherwise equivalent.

The colon (:) and equals sign (=) symbols together are called the assignment operator. When used as the assignment operator, there can be no spaces or comments between the two symbols.

## Structured Statements

A structured statement contains one or more statements that are called, collectively, a statement list. The structured statement determines when the statement list it contains will be executed.

There are four structured statements:

| Statement | Description |
|---|---|
| BEGIN | Provides a logical grouping of statements that performs a specific function. |
| FOR | Executes a list of statements while a variable is incremented or decremented from an initial value to a final value. |
| REPEAT | Executes a list of statements until a specified condition is true. The test is made after each execution of the statements. |
| WHILE | Executes a list of statements while a specified condition is true. The test is made before each execution of the statements. |

The IF and CASE control statements (described later in this chapter) also contain statement lists. The structured statements, the IF statement, and the CASE statement can be nested within each other up to 63 levels. The FOR statement can be nested 15 levels.

## BEGIN Statement

The BEGIN statement executes a single statement list once; there is no repetition. This statement logically groups statements that perform a particular function and improves readability.

Use this format for the BEGIN statement:

> *{/label/}*
> **BEGIN**
>     **statement list;**
> **END** *{/label/}* **;**

*label*

Name that identifies the BEGIN statement and the statement list contained in it. Use of labels is optional. If you use a label before BEGIN, it is recommended that you use one after END, but it is not required. If you use labels in both places, they must match. The label name must be unique within the block in which you use it.

**statement list**

One or more statements.

Declarations are not allowed with the BEGIN statement. Execution of the BEGIN statement ends when either the last statement in the list is executed or control is explicitly transferred from within the list.

## FOR Statement

The FOR statement executes a statement list repeatedly while a special variable ranges from an initial value to a final value. There are two formats for the FOR statement: one that increments the variable and one that decrements the variable.

Use this format to increment the variable:

> *{/label/}*
> **FOR name := initial_value TO final_value DO**
> **statement list;**
> **FOREND** *{/label/}* **;**

Use this format to decrement the variable:

> *{/label/}*
> **FOR name := initial_value DOWNTO final_value DO**
> **statement list;**
> **FOREND** *{/label/}* **;**

*label*

Name that identifies the FOR statement and the statement list contained in it. Use of labels is optional. If you use a label before FOR, it is recommended that you use one after FOREND, but it is not required. If you use labels in both places, they must match. The label name must be unique within the block in which you use it.

**name**

Name of the variable that controls the number of repetitions of the statement list. This variable keeps track of the number of iterations performed or the current position within the range of values.

**initial_value**

Scalar expression specifying the initial value assigned to the variable.

**final_value**

Scalar expression specifying the final value to be assigned to the variable if the statement ends normally. If the statement ends abnormally or as the result of an EXIT statement, this may not be the actual final value.

**statement list**

One or more statements.

The variable, initial value, and final value must be of equivalent scalar types or subranges of equivalent types. The variable cannot be assigned a value within the statement list, or be passed as a reference parameter to a procedure called within the statement list. Either condition causes a fatal compilation error. The variable cannot be an unaligned component of a packed structure.

When CYBIL encounters a FOR statement that increments (one containing the TO clause), it evaluates the initial value and final value. If the initial value is greater than the final value, the FOR statement ends and execution continues with the statement following FOREND; the statement list is not executed. If the initial value is less than or equal to the final value, the initial value is assigned to the control variable and the statement list is executed. Then, the control variable is incremented by one value and, for each increment, the statement list is executed. This sequence of actions continues through the final value. For example, the statement

```
FOR i = 1 TO 5 DO
  :
FOREND;
```

causes the statement list to be executed five times, that is, while I takes on values from 1 to 5. Then the FOR statement ends. Execution continues at the statement following FOREND with the variable I having a value of 5.

When CYBIL encounters a FOR statement that decrements (one containing the DOWNTO clause), it performs a similar process. If the initial value is less than the final value, the FOR statement ends and execution continues with the statement following FOREND. If the initial value is greater than or equal to the final value, the initial value is assigned to the control variable and the statement list is executed. The control variable is decremented by one value and, for each decrement, the statement list is executed. When the control variable reaches the final value and the statement list is executed the last time, the FOR statement ends.

The initial value and final value expressions are evaluated once, when the statement is entered; the values are then held in temporary locations. Thus, subsequent assignments to initial value and final value have no effect on the execution of the FOR statement.

When a FOR statement completes normally, the value of the control variable is that of the final value specified in the statement. This may not be the case if the statement ends abnormally or ends as a result of an EXIT statement.

FOR statements can be nested in up to 15 levels.

Example:

Integer values are often used in FOR statements, but any scalar type can be used. The following example executes a statement list while the value of a character variable is incremented.

```
FOR control := 'a' TO 'z' DO
  :
FOREND;
```

Each time the statement list is performed, the value of CONTROL increases by one value, following the normal sequence of alphabetic characters from 'a' to 'z'; that is, after the statement list is executed once, the value of CONTROL changes to 'b', and so on until the statement list has been executed 26 times.

## REPEAT Statement

The REPEAT statement executes a statement list repeatedly until a specific condition is true.

Use this format for the REPEAT statement:

> *{/label/}*
> **REPEAT**
>     **statement list;**
> **UNTIL expression;**

*label*

Name that identifies the REPEAT statement and the statement list contained in it. Use of the label before REPEAT is optional; a label is not permitted after UNTIL. The label name must be unique within the block in which it is used.

**statement list**

One or more statements.

**expression**

A boolean type expression.

The statement list is always executed at least once. After the last statement in the list, the expression is evaluated. Every time the expression is FALSE, the statement list is executed again. When the expression is TRUE, the REPEAT statement ends and execution continues with the statement following the UNTIL clause.

The statement list can contain nested REPEAT statements.

Example:

In this example, the statement list (mod operation and assignments) is executed once. If J is not equal to zero, it is executed again and continues until J is equal to zero.

```
REPEAT
  k := i MOD j;
  i := j;
  j := k;
UNTIL j = 0;
```

## WHILE Statement

The WHILE statement executes a statement list repeatedly while a specific condition is true.

Use this format for the WHILE statement:

> *{/label/}*
> **WHILE expression DO**
>     **statement list;**
> **WHILEND** *{/label/}* **;**

*label*

Name that identifies the WHILE statement and the statement list contained in it. Use of labels is optional. If you use a label before WHILE, it is recommended that you use one after WHILEND, but it is not required. If you use labels in both places, they must match. The label name must be unique within the block in which you use it.

**expression**

A boolean type expression.

**statement list**

One or more statements.

If the boolean expression is evaluated as TRUE, the statement list is executed. After the last statement in the list, the expression is again evaluated. Every time the expression is TRUE, the statement list is executed. When the expression is FALSE, the WHILE statement ends and execution continues with the statement following WHILEND. If the expression is FALSE in the initial evaluation, the statement list is never executed.

Example:

In this example, the expression TABLE[I] < > 0 is evaluated; an element of the array TABLE is compared to 0. While the expression is true (the element is not 0), I is incremented. This causes the next element of the array to be checked. When the expression is false, the statement list is not executed. Execution continues with the statement following WHILEND. I is the position of an element in the array that is 0.

```
/check_for_zero/
  WHILE table [i] <> 0 DO
    i := i + 1;
  WHILEND /check_for_zero/;
```

The preceding example assumes, of course, that the array contains an element with the value 0. If not, the WHILE statement list executes in an infinite loop. In either the WHILE expression or the statement list, there must be a check. One solution is to set a variable, TABLE_MAX, to the maximum number of elements in the array and check it before executing the statement list, as in:

```
WHILE (i < table_max) AND (table [i] <> 0) DO
```

Now both expressions must be true before the statement list is executed. If either is false, execution continues following WHILEND.

## Control Statements

A control statement can change the flow of execution of a program by transferring control from one place in the program to another.

There are five control statements:

| Statement | Description |
|---|---|
| IF | Executes one statement list if a given condition is true; ends the statement or executes another statement list if the condition is false. |
| CASE | Executes one statement list out of a set of statement lists, depending on the value of a given expression. |
| CYCLE | Causes the remaining statements in a repetitive statement (FOR, REPEAT, or WHILE) to be skipped and the next iteration of the statement to occur. |
| EXIT | Unconditionally stops execution within a procedure, function, or a structured statement (BEGIN, REPEAT, WHILE, and FOR). |
| RETURN | Returns control from a procedure or function to the point at which it was called. |

The structured statements (described earlier in this chapter) contain statement lists like the IF and CASE statements. The structured statements, the IF statement, and the CASE statement can be nested within each other up to 63 levels.

Procedure and function calls also transfer control of an executing program. Functions are discussed in chapter 6 and procedures are discussed in chapter 7.

## IF Statement

The IF statement executes or skips a statement list, depending on whether a given condition is true or false.

Use this format for the IF statement:

**IF expression THEN**
**statement list;**
*{ELSEIF expression THEN*
*statement list;}...*
*{ELSE*
*statement list;}*
**IFEND;**

**expression**

A boolean expression.

**statement list**

One or more statements.

The ELSEIF and ELSE clauses are optional. The ELSEIF clause contains another test condition that is evaluated only if the preceding condition (expression) is false. The ELSE clause provides a statement list that is executed unconditionally when the preceding expression is false.

When an expression is evaluated as true, the statement list following the reserved word THEN is executed. When the list is completed, execution continues with the first statement following IFEND. If the expression is false, execution continues with the next clause or reserved word in the IF statement format (that is, ELSEIF, ELSE, or IFEND).

If the next reserved word in the IF statement format is IFEND, execution continues with the first statement following it.

If the next reserved word is ELSEIF, the expression contained in that clause is evaluated; if true, the statement list that follows is executed. Otherwise, execution continues with the next reserved word in the IF statement format.

If the next reserved word is ELSE, the statement list that follows is always executed. You get to this point only if the preceding expression(s) is false.

Additional IF statements can be contained (nested) in any of the statement lists. A consistent style of indentation or spacing, such as that provided by the CYBIL source code formatter, greatly improves the readability of such statements. (The source code formatter is described in chapter 8.)

If the ELSE clause is included in a nested IF statement, the clause applies to the most recent IF statement.

Examples:

In this example, Y is assigned to X only if X is less than Y.

```
IF x < y THEN
  x := y;
IFEND;
```

In the next example, Z is always assigned one of the values 1, 2, 3, or 4 depending on the value of X.

```
IF x <= 5 THEN
  z := 1;
ELSEIF x > 30 THEN
  z := 2;
ELSEIF x = 15 THEN
  z := 3;
ELSE
  z := 4;
IFEND;
```

## CASE Statement

The CASE statement executes one statement list out of a set of lists, based on the value of a given expression.

Use this format for the CASE statement:

> **CASE expression OF**
> = **value** *{,value}...* =
>    **statement list;**
> *{= value {,value}... =*
>    *statement list;}...*
> *{ELSE statement list;}*
> **CASEND;**

**expression**

A scalar expression. The expression must be of the same type as the value or values that follow.

**value**

One or more constant scalar expressions or a subrange of constant scalar expressions. A subrange indicates that all of the values included in the subrange are acceptable values. If you specify two or more values, separate them with commas. The values must be of the same type as the expression. Values can be in any order, not strictly sequential. Values must be unique within the CASE statement.

**statement list**

One or more statements.

You define a set of possible values that a variable or expression can have. With one or more of the values you associate a statement list using the format:

```
= value =
   statement list;
```

When the CASE statement is executed, the expression is evaluated and the statement list associated with the current value of the expression is executed. If the current value is not found among those in the CASE statement, execution continues with the ELSE clause. If ELSE is omitted and the value is not found in the CASE statement, the program is in error. After any one of the statement lists is executed, execution continues with the statement following CASEND.

Examples:

In this example, I is a variable that is expected to take on one of the values -5 through 20. If its value is -5, -4, -3, -2, -1, or 0, the first statement list (X := X) is executed and control goes to the statement following CASEND. If the value of I is 1, the second statement list is executed, and so on.

```
CASE i OF
= -5, -4, -3, -2, -1, 0 =
  x := x;
= 1 =
  x := x + 1;
= 2 =
  x := x + 2;
= 3 =
  x := x + 3;
= 4 =
  x := x + 4;
= 5 .. 20 =
  x := x + 5;
CASEND;
```

In the next example, OPERATOR is a variable that is expected to take on values of PLUS, MINUS, or TIMES. Depending on the current value of OPERATOR, the associated statement is executed.

```
CASE operator OF
= plus =
  x := x + y;
= minus =
  x := x - y;
= times =
  x := x * y;
ELSE
  x := 0;
  y := 0;
CASEND;
```

## CYCLE Statement

The CYCLE statement can be included in the statement list of a repetitive statement (FOR, REPEAT, or WHILE) and causes any statements following it to be skipped and the next iteration of the repetitive statement to occur.

Use this format for the CYCLE statement:

**CYCLE /label/;**

**label**

Name that identifies the repetitive statement in which the CYCLE statement is contained.

The CYCLE statement is usually used in conjunction with an IF statement, as in:

```
/label/
repetitive statement
    IF expression THEN
    CYCLE /label/;
    IFEND;
    remainder of statement list;
end of repetitive statement;
```

The IF statement tests for a condition that, if true, causes the CYCLE statement to be executed. Then the remaining statements of the repetitive statement are skipped and execution continues with whatever would normally follow the statement list, either another cycle of the repetitive statement or the next statement following the end of the repetitive statement. If the condition in the IF statement is false, the remaining statements in the repetitive statement are executed.

If it is not contained in a repetitive statement, the CYCLE statement is diagnosed as a compilation error.

Example:

This example finds the smallest element of an array TABLE. On the first execution, X (the first element of the array) is assumed to be smallest. If X is smaller than succeeding elements of the array, the CYCLE statement is executed; the remainder of the statements are then skipped, and the next iteration of the FOR statement occurs. If an element smaller than X is found, the CYCLE statement is ignored and the rest of the statement list is processed; X is replaced by the smaller element. If N has not yet been reached, the FOR statement continues. When N is reached, X will contain the smallest element of the array.

```
x := table [1];

/find_smallest/
  FOR k := 2 TO n DO
    IF x < table [k] THEN
      CYCLE /find_smallest/;
    IFEND;
    x := table [k];
  FOREND /find_smallest/;
```

## EXIT Statement

The EXIT statement causes an unconditional exit from a procedure, function, or a structured statement (BEGIN, FOR, REPEAT, and WHILE).

Use this format for the EXIT statement:

**EXIT name;**

**name**

Name that identifies the procedure, function, or statement. For a procedure or function, it is the procedure or function name. For a structured statement, it is the statement label; in this case the format is:

**EXIT /label/;**

When the EXIT statement is encountered, execution of the named procedure, function, or statement is automatically stopped and execution resumes with the statement that would follow normal completion. For a procedure or function, it is the statement that would normally follow the procedure or function call. For a structured statement, it is the statement following the end of the structured statement (END, FOREND, UNTIL expression, and WHILEND).

The EXIT statement must be within the scope of the procedure, function, or statement it names. Otherwise, it has no meaning and is diagnosed as a compilation error.

With a single EXIT statement, you can exit several levels of procedures, functions, or statements; they need not be exited separately. (This is sometimes referred to as a nonlocal exit.) If the EXIT statement is executed in a nested recursive procedure or function, it is the most recent invocation of the procedure or function and any intervening procedures or functions that are exited.

Example:

The following example declares an array of user names, then sets the variable KEY to one of the names. The statement list in the FOR statement labeled FIND_KEY searches the array for the key name. When it is found, the EXIT statement is executed and the FOR statement ends. Execution continues at the statement following the end of the FOR statement which is the end of the procedure.

```
PROCEDURE exit_example;

  VAR
    i: integer,
    key: string(7),
    names: [READ] array [ 1 .. 4 ] of string(7) :=['jqp8402',
          'jxd1432', 'efd3204', 'led4411'];

  key := 'efd3204';

  /find_key/
  FOR i := LOWERBOUND (names) TO UPPERBOUND (names) DO
    IF key = names[i] THEN
      EXIT /find_key/;
    IFEND;
  FOREND/find_key/;
PROCEND exit_example;
```

## RETURN Statement

The RETURN statement completes the execution of a procedure or function and returns control to the program, procedure, or function that called it.

Use this format for the RETURN statement:

**RETURN;**

If it is omitted at the end of a procedure or function, the RETURN statement is assumed.

## Storage Management Statements

Storage management statements allow you to manipulate components of sequence and heap types, and put variables in the run-time stack.

There are five storage management statements:

| Statement | Description |
|---|---|
| RESET | Resets the pointer in a sequence or releases all the variables in a user-defined heap. |
| NEXT | Creates or accesses the next element of a sequence given a starting element. |
| ALLOCATE | Allocates storage for a variable in a heap. |
| FREE | Releases a variable from a heap. |
| PUSH | Allocates storage for a variable in the run-time stack. |

Sequences use the RESET and NEXT statements. Heaps use the RESET, ALLOCATE, and FREE statements. The run-time stack uses the PUSH statement. (Refer to Storage Types in chapter 4 for further information on sequences and heaps.) The NEXT and ALLOCATE statements can also be used to allocate space in a segment access file. Accessing a file as a memory segment is described in the CYBIL Sequential and Byte-Addressable Files manual. That manual also compares use of the default heap and run-time stack with use of a segment access file for data storage.

In the NEXT, ALLOCATE, and PUSH statements, you must specify a pointer to the variable to be manipulated so that sufficient space can be allocated for that type. This pointer can be a pointer to a fixed type, a pointer to an adaptable type, or a pointer to a bound variant record type. Space is then allocated for a variable of the type to which the pointer can point. This pointer is also used to access the variable. When space is allocated, CYBIL returns the address of the variable to the pointer. Therefore, to reference a variable in a sequence, heap, or the run-time stack, you indicate the object of the pointer in this form: pointer name ^.

If you specify a fixed type pointer, the statement uses a variable of the type designated by that pointer variable. If you specify an adaptable type pointer or bound variant record type pointer, you must also indicate the size of the adaptable type or the tag field of the variant record to be used. This causes a fixed type to be set and the adaptable or bound variant record pointer designates a variable of that fixed type. That particular fixed type is designated until it is reset by a subsequent assignment or another storage management statement.

To indicate the size of an adaptable pointer or the tag field of a bound variant record pointer, you use the format:

**pointer : [size fixer]**

**pointer**

Name of an adaptable pointer variable or a bound variant record pointer variable.

**size fixer**

Fixed amount of space required for the variable designated by pointer. You set the size of the adaptable type the same way you specify the size of the corresponding unadaptable (fixed) type. For example, in a variable or type declaration, you specify the size of a fixed array with subscript bounds, usually a subrange of "scalar expression..scalar expression". You set the size of an adaptable array here using the same form. Summarized next are the forms used to set the size of all possible adaptable types. For more detailed information, refer to the descriptions of the corresponding fixed types in chapter 4.

| **Pointer Type** | **Form Used to Set Size** |
|---|---|
| Adaptable array | scalar expression .. scalar expression |
| Adaptable string | A positive integer expression specifying the length of the string |
| Adaptable heap | [*{REP positive integer expression OF}* fixed type name *{,{REP positive integer expression OF} fixed type name}...*] |
| Adaptable sequence | [*{REP positive integer expression OF}* fixed type name *{,{REP positive integer expression OF} fixed type name}...*] |
| Adaptable record | One of the forms used for an adaptable array, string, heap, or sequence |
| Bound variant record | A scalar expression or one or more constant scalar expressions followed by an optional scalar expression |

If an adaptable array had a lower bound specified in its original declaration, the lower bound specified here must match that value. For an adaptable record, the form used must be a value and type to which the record can adapt. For a bound variant record, the order, types, and values used must be valid for a variant of the record; all but the last of the expressions must be constant.

Examples:

This example declares a type that is an adaptable array named ADAPT_ARRAY. PTR is a pointer to that type. BUNCH is a heap with space for 100 integers. The heap BUNCH is reset; that is, any existing elements are released. Space is then allocated in the heap for a variable of the type designated by PTR. That variable is of type ADAPT_ARRAY (an array of integers) and it has fixed subscript bounds of from 1 to 15. PTR now points to that array.

```
TYPE
  adapt_array = array [1 .. * ] of integer;

VAR
  ptr: ^adapt_array,
  bunch: HEAP (REP 100 of integer);

RESET bunch;
ALLOCATE ptr: [1 .. 15] IN bunch;
```

The following example shows the setting of an adaptable sequence. Notice that two sets of brackets are required in the PUSH statement.

```
VAR
  ptr: ^SEQ ( * );

PUSH ptr: [[REP 10 OF integer, REP 22 OF char]];
```

## RESET Statement

The RESET statement operates on both sequences and heaps. In a sequence, it resets the pointer to the beginning of the sequence or to a specific variable within the sequence. In a heap, it releases all the variables in the heap.

The RESET statement must appear before the first NEXT statement (for a sequence) or ALLOCATE statement (for a user-defined heap). This ensures that the sequence is at the beginning or the heap is empty. If you reserve space by using a NEXT or ALLOCATE statement before the RESET statement, the program is in error.

### *RESET in a Sequence*

This statement sets the current element being pointed to in a sequence.

Use this format for the RESET statement in a sequence:

**RESET sequence_pointer** *{ TO variable_pointer }* **;**

**sequence_pointer**

Name of a pointer to a sequence. This specifies the particular sequence.

*variable_pointer*

Name of a pointer to a particular variable within the sequence. If it is omitted, the pointer points to the first element of the sequence.

If you did not set the value of the variable_pointer with a NEXT statement for the same sequence, an error will occur. An error will also occur if the value of the variable_pointer is NIL.

The RESET statement must appear before the first occurrence of a NEXT statement to reset the sequence to its beginning; otherwise, the program is in error.

*RESET in a Heap*

This statement releases the variables currently in a heap.

Use this format for the RESET statement in a heap:

**RESET heap;**

**heap**

Name of a heap type variable.

Space for the variables is released and their values become undefined.

Make sure that the RESET statement appears before the first occurrence of an ALLOCATE statement for a user-defined heap so that the heap is empty; otherwise, the program is in error.

## NEXT Statement

The NEXT statement sets the specified pointer to designate the current element of the sequence and then makes the next element in the sequence the current element. This moves the pointer along the sequence, allowing you to assign values to and access elements.

Use this format for the NEXT statement:

**NEXT pointer** *{ : [size fixer] }* **IN sequence_pointer;**

**pointer**

Name of a pointer to a fixed type, an adaptable type, or a bound variant record type. The type pointed to by the pointer is the type of the variable in the sequence. These pointers are described in detail under Storage Management Statements earlier in this section.

*size fixer*

Size of an adaptable type or tag field of a bound variant record type. If it is omitted, the pointer must be a pointer to a fixed type. The forms used to specify size are described in detail under Storage Management Statements earlier in this section.

**sequence_pointer**

Name of a pointer to a sequence. This specifies the particular sequence.

After a RESET statement, the current element is always the first element of the sequence. A NEXT statement assigns to the specified pointer the address of the current (first) element, and then makes the next element (the second) the new current element. Thus, the order of variables in a sequence is determined by the order in which the NEXT statements are executed.

If the NEXT statement causes the new element to be outside the bounds of the sequence, the pointer is set to NIL. Before attempting to reference an element in a sequence, check for a NIL pointer value. If you use a pointer variable with a value of NIL to access an element, a run-time error will occur.

The type of pointer you specify when data is retrieved from the sequence must be equivalent to the type you used when the same data was stored in the sequence; otherwise, the program is in error.

## ALLOCATE Statement

The ALLOCATE statement allocates storage space for a variable of the specified type in the specified heap and then sets the pointer to point to that variable.

Use this format for the ALLOCATE statement:

**ALLOCATE pointer** *{ : [size fixer] } { IN heap }* **;**

**pointer**

Name of a pointer to a fixed type, adaptable type, or bound variant record type. These pointers are described in detail under Storage Management Statements earlier in this section.

*size fixer*

Size of an adaptable type or tag field of a bound variant record type. If it is omitted, the pointer must be a pointer to a fixed type. The forms used to specify size are described in detail under Storage Management Statements earlier in this section.

*heap*

Name of a heap type variable. If it is omitted, the default heap is assumed.

If there is not enough space for the variable to be allocated, the pointer is set to NIL. Before attempting to reference a variable in a heap, check for a NIL pointer value. If you use a pointer variable with a value of NIL to access data, a run-time error will occur.

For a user-defined heap, you must include a RESET statement before the first occurrence of an ALLOCATE statement to ensure that the heap is empty; otherwise, the program is in error. (When you use the default heap, however, the RESET statement is done automatically and you should not specify it.)

The lifetime of a variable that is allocated using the storage management statements is the time between the allocation of storage (with the ALLOCATE statement) and the release of storage (with the FREE statement). A variable allocated using an automatic pointer must be explicitly freed (using the FREE statement) before the block is left, or the space will not be released by the program. When the block is left, the pointer no longer exists and, therefore, the variable cannot be referenced. If the block is entered again, the previous pointer and the variable referenced by the pointer cannot be reclaimed. Therefore, it is recommended that you free such variables before leaving the block.

## FREE Statement

The FREE statement releases the specified variable from the specified heap.

Use this format for the FREE statement:

**FREE pointer** *{ IN heap }* **;**

**pointer**

Name of the pointer variable that designates the variable to be released.

*heap*

Name of a heap type variable. If it is omitted, the default heap is assumed.

The variable's space in the heap is released and its value becomes undefined. The pointer variable designating the released variable is set to NIL. If you specify a variable that is not currently allocated in the heap, the results are unpredictable.

Using a pointer variable with the value NIL to access data causes a run-time error to occur. Releasing the NIL pointer is also an error.

## PUSH Statement

The PUSH statement allocates storage space on the run-time stack for a variable of the specified type and then sets the pointer to point to that variable.

Use this format for the PUSH statement:

**PUSH pointer** *{ : [size fixer] }* **;**

**pointer**

Name of a pointer to a fixed type, adaptable type, or bound variant record type. These pointers are described in detail under Storage Management Statements earlier in this section.

*size fixer*

Size of an adaptable type or tag field of a bound variant record type. If it is omitted, the pointer must be a pointer to a fixed type. The forms used to specify size are described in detail under Storage Management Statements earlier in this section.

If there is not enough space for the variable to be allocated, the pointer is set to NIL. The value of the variable that has just been allocated is undefined until a subsequent assignment to the variable is made.

You cannot release space on the run-time stack explicitly. It is released automatically when the procedure containing the PUSH statement completes. At that time, space for the variable is released and its value becomes undefined.

Example:

This example shows the declaration of a pointer variable named ARRAY_PTR that points to an adaptable array. The PUSH statement allocates space in the run-time stack for a fixed array of from 1 to 20 elements. Elements of the array can be referenced by ARRAY_PTR^[i], where i is an integer from 1 to 20.

```
VAR
  array_ptr: ^array [1 .. * ] of integer;
PUSH array_ptr: [1 .. 20];
```

# Functions 6

This chapter describes the functions that are predefined in CYBIL and explains how to define your own functions.

# Functions 6

A function is one or more statements that perform a specific action and can be called by name from a statement elsewhere in a program. A reference to a function causes actual parameters in the calling statement to be substituted for the formal parameters in the function declaration and then the function's statements to be executed. Usually the function computes a value and returns it to the portion of the program that called it.

A function differs from a procedure in that the value returned for a function replaces the actual function reference within the statement. A function is a valid operand in an expression; the value returned by the function replaces the reference and becomes the operand.

The value of a function is the last value assigned to it before the function returns to the point where it was called. The reason for its return doesn't matter; it could complete normally or abnormally. If the function returns for any reason before a value is assigned to the function name, results are undefined.

Functions can be recursive; that is, a function can call itself. In that case, however, there must be some provision for ending the calls and the code within the function must not modify itself. Appendix F, The CYBIL Run-Time Environment, describes how recursive functions are managed.

You can call standard functions that are already defined in the CYBIL language, you can define your own functions, or you can call functions designed specifically for use on NOS/VE. This chapter describes all three.

Functions that start with $ are data conversion functions. Functions that start with # are either system-dependent functions (that is, unique to CYBIL on NOS/VE) or functions whose results are system dependent. (For example, #SIZE is a standard function available on all variations of CYBIL regardless of operating system; however, its results vary depending on the system on which it is being used.)

## Standard Functions

The functions described here are standard CYBIL functions. They can be used safely in variations of CYBIL available on other operating systems. Under System-Dependent Functions, later in this chapter, you'll find descriptions of functions unique to CYBIL on NOS/VE.

The functions are described in alphabetical order.

### $CHAR Function

The $CHAR function returns the character whose ordinal number within the ASCII collating sequence is that of a given expression.

Use this format for the $CHAR function call:

**$CHAR(expression)**

**expression**

An integer expression whose value can be from 0 to 255.

If you specify a value for the integer expression less than 0 or greater than 255, an error occurs.

## $INTEGER Function

The $INTEGER function returns the integer value of a given expression.

Use this format for the $INTEGER function call:

**$INTEGER(expression)**

**expression**

An expression of type integer, subrange of integer, boolean, character, ordinal, or real.

| Expression | Value Returned |
|---|---|
| Integer | The value of that expression is returned. |
| Boolean | Zero is returned for a false expression and 1 is returned for a true expression. |
| Character | The ordinal number of the character in the ASCII collating sequence is returned. |
| Ordinal | The ordinal number associated with that ordinal value is returned. The value returned for the first element of an ordinal type is 0, the second element is 1, and so on. |
| Real | The value of the expression is truncated to a whole number. If the number is in the range defined for integers, that number is returned; otherwise, an out-of-range error occurs. |

## #LOC Function

The #LOC function returns a pointer to the first cell allocated for a given variable.

Use this format for the #LOC function call:

**#LOC(name)**

**name**

Name of a variable.

If the specified variable is a formal value parameter, the pointer cannot be used to modify the variable.

## LOWERBOUND Function

The LOWERBOUND function returns the lower bound of an array's subscript bounds.

Use this format for the LOWERBOUND function call:

**LOWERBOUND(array)**

**array**

An array variable or the name of a fixed array type.

The type of the value returned is the same as the type of the array's subscript bounds.

Example:

Assuming the following declaration has been made

```
VAR
  x: array [1 .. 100] of boolean,
  y: array ['a' .. 't'] of integer;
```

the value of LOWERBOUND(X) is 1; the value of LOWERBOUND(Y) is 'a'.

## LOWERVALUE Function

The LOWERVALUE function returns the smallest possible value that a given variable or type can have.

Use this format for the LOWERVALUE function call:

**LOWERVALUE(name)**

**name**

A scalar variable or name of a scalar type.

The type of the value returned is the same as the given type.

Examples:

Assuming the following declaration has been made

```
VAR
  dozen: 1 .. 12;
```

the value of LOWERVALUE(DOZEN) is 1.

After the declarations

```
TYPE
  t = (first, second, third);

VAR
  v: t;
```

the value of LOWERVALUE(V) is FIRST and the value of LOWERVALUE(T) is FIRST.

## PRED Function

The PRED function returns the predecessor of a given expression.

Use this format for the PRED function call:

**PRED(expression)**

**expression**

A scalar expression.

If the predecessor of the expression does not exist, the program is in error.

Example:

The following example declares two variables, WARM and COLD, each of which can take on ordinal values of the type SEASONS. The variable WARM is assigned the value SPRING while the variable COLD is assigned the value WINTER.

```
TYPE
  seasons = (winter, spring, summer, fall);

VAR
  warm: seasons,
  cold: seasons;

warm := spring;
cold := PRED (warm);
```

## #PTR Function

The #PTR function returns a pointer that can be used to access the object of a relative pointer.

Use this format for the #PTR function call:

**#PTR(pointer_name** *{,parent_name}* )

**pointer_name**

Name of the relative pointer variable.

*parent_name*

Name of the variable that contains the components being designated by relative pointers. If omitted, the default heap is used. The variable can be a string, array, record, heap, or sequence type (either fixed or adaptable).

Relative pointers cannot be used to access data directly. The #PTR function converts a relative pointer to a pointer in order to reference the object of the relative pointer.

The type of the object pointed to by the returned pointer is the same as the type of the object pointed to by the relative pointer. If the type of the parent variable associated with the specified relative pointer is not equivalent to the type of the specified parent variable, a compile-time error occurs.

For further information on relative pointers, refer to Pointer Types in chapter 4.

## $REAL Function

The $REAL function returns the real number equivalent of a given integer expression.

Use this format for the $REAL function call:

**$REAL(expression)**

**expression**

An integer expression.

## #REL Function

The #REL function returns a relative pointer.

Use this format for the #REL function call:

**#REL(pointer_name** *{,parent_name}* )

**pointer_name**

Name of the direct pointer variable.

*parent_name*

Name of the variable that contains the components being designated by relative pointers. If omitted, the default heap is used. The variable can be a string, array, record, heap, or sequence type (either fixed or adaptable).

The type of the relative pointer's object is the same as the type of the given direct pointer's object. (This type was specified in the VAR declaration of the relative pointer variable.) The parent type of the relative pointer's object is the same as the type of the specified parent variable.

If the pointer specified in the function call does not designate an element of the parent variable, the result is undefined.

Relative pointer values can be generated solely through this function. For further information on relative pointers, refer to Pointer Types in chapter 4.

## #SEQ Function

The #SEQ function returns an adaptable pointer to a sequence allocated for a given variable.

Use this format for the #SEQ function call:

**#SEQ(name)**

**name**

Name of a variable of any type.

The following relationships hold between the #LOC, #SEQ, and #SIZE functions:

```
#LOC(#SEQ(name) ^ ) = #LOC(name)

#SIZE(#SEQ(name) ^ ) = #SIZE(name)
```

## #SIZE Function

The #SIZE function returns the number of cells required to contain a given variable or a variable of a specified type.

Use this format for the #SIZE function call:

**#SIZE(name)**

**name**

Name of a variable, fixed record type, bound variant record, or an adaptable type.

If you specify the name of a bound variant record type, the variant that requires the largest size is used. If you specify the name of an adaptable type, you must also supply a size fixer for the type.

Example:

The following example declares a procedure, FIND_SIZE, that has as its only parameter an adaptable array named A. An adaptable array type named B is also declared inside the procedure. When the procedure is called, the first #SIZE function determines the size of array A, the fixed array that was passed to it from the caller. The second #SIZE function determines the size of array B using a size fixer (the subrange is 1 to 100).

```
PROCEDURE find_size (a: array [1 .. *] OF integer);
  VAR
    i: integer;

  TYPE
    b= array [1 .. *] of integer;

  i := #SIZE(a);

  i := #SIZE(b: [1 .. 100]);
PROCEND find_size;
```

## STRLENGTH Function

The STRLENGTH function returns the length of a given string.

Use this format for the STRLENGTH function call:

**STRLENGTH(string)**

**string**

A string variable, name of a string type, or adaptable string reference.

For a fixed string, the allocated length is returned as an integer subrange. For an adaptable string, the current length is returned.

Example:

The following example declares a procedure, FIND_LENGTH, that has as its only parameter an adaptable string named S. When the procedure is called, the STRLENGTH function determines the length of the fixed string that was passed to it.

```
PROCEDURE find_length (s: string(*));
  VAR i: integer;
    :
  i := STRLENGTH (s);
```

## SUCC Function

The SUCC function returns the successor of a given expression.

Use this format for the SUCC function call:

**SUCC(expression)**

**expression**

A scalar expression.

If the successor of the expression does not exist, the program is in error.

Example:

The following example declares two variables, HOT and COOL, each of which can take on ordinal values of the type SEASONS. The variable HOT is assigned the value SUMMER while the variable COOL is assigned the value FALL.

```
TYPE
  seasons = (winter, spring, summer, fall);

VAR
  hot: seasons,
  cool: seasons;

hot := summer;
cool := SUCC (hot);
```

## UPPERBOUND Function

The UPPERBOUND function returns the upper bound of an array's subscript bounds.

Use this format for the UPPERBOUND function call:

**UPPERBOUND(array)**

**array**

An array variable or the name of a fixed array type.

The type of the value returned is the same as the type of the array's subscript bounds.

Examples:

Assuming the following declaration has been made

```
VAR
  x: array [1 .. 100] of boolean,
  y: array ['a' .. 't'] of integer;
```

the value of UPPERBOUND(X) is 100; the value of UPPERBOUND(Y) is 't'.

In the following example, the value of UPPERBOUND(TABLE) is 50:

```
VAR
  table: ^array [1 .. * ] of cell;

ALLOCATE table: [1 .. 50];
```

## UPPERVALUE Function

The UPPERVALUE function returns the largest possible value that a given variable or type can have.

Use this format for the UPPERVALUE function call:

**UPPERVALUE(name)**

**name**

A scalar variable or name of a scalar type.

The type of the value returned is the same as the given type.

Examples:

Assuming the following declaration has been made

```
VAR
  dozen: 1 .. 12;
```

the value of UPPERVALUE(DOZEN) is 12.

After the declarations

```
TYPE
  t = (first, second, third);

VAR
  v: t;
```

the value of UPPERVALUE(V) is THIRD and the value of UPPERVALUE(T) is THIRD.

## User-Defined Functions

This section describes how you define your own functions.

### Function Declaration

You define your own function with the function declaration.

Use this format to declare a function:

> **FUNCTION** *{[attributes]}* **name** *{(formal_parameters)}* **:**
>     **result_type;**[1]
>         *{declaration_list;}*
>         **statement_list;**
> **FUNCEND** *{name}* **;**

*attributes*

One or more of the following attributes. If you specify more than one, separate them with commas.

XREF

The function has been compiled in a different module. In this case, the function declaration can contain the name and formal parameters, but no declaration list or statement list. In the other module, the function must have been declared with the XDCL attribute and an identical parameter list. If omitted, the function must be defined within the module where it is called.

XDCL

The function can be called from outside of the module in which it is located. This attribute can be included only in a function declared at the outermost level of a module; it cannot be contained in a program, procedure, or another function. Other modules that call this function must contain the same function declaration with the XREF attribute specified.

1. Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a function declaration. If included in a CYBIL program run on NOS/VE, this parameter is ignored.

INLINE

Instead of calling the function, the compiler inserts the actual function statements at the point in the code where the function call is made. Additional information on inline functions is given later in this section.

#GATE[2]

The function can be called by a function call from a higher ring level if the call is issued from within the call bracket of the gated function.[3] If you specify #GATE, you must also specify the XDCL attribute.

If you don't specify any attributes, the function is assumed to be in the same module in which it is called.

**name**

Name of the function. The function name is optional following FUNCEND.

*formal_parameters*

One or more parameters in the form:

**VAR name** *{,name}...* **: type**
*{,name {,name}... : type}...*

and/or:

**name** *{,name}...* **: type**
*{,name {,name}... : type}...*

The first form is called a reference parameter; the second form is called a value parameter. There is essentially no difference between them in the context of a function. However, procedures (and programs) do treat them differently. Both kinds of parameters can appear in the formal parameter list; if so, they are separated by semicolons (for example, [I:INTEGER; VAR A:CHAR). Reference and value parameters are discussed in more detail later in this chapter under Parameter List.

The maximum number of parameters that can be passed to an externally referenced (XREF) function is 126.

2. This attribute is not supported on variations of CYBIL available on other operating systems.

3. A ring level is a hardware feature. Rings provide hardware protection in that an unauthorized program cannot access anything at a lower ring level. For further information on rings, refer to the SCL Object Code Management manual.

**result_type**

The type of the result to be returned. Specify any fixed scalar, floating-point, pointer, or cell type.

*declaration_list*

Zero or more declarations.

**statement_list**

One or more statements.

In an assignment statement within a function, the lefthand side of the statement (the variable to receive the value) cannot be:

- A nonlocal variable.
- A formal parameter of the function.
- The object of a pointer variable.

User-defined functions cannot contain:

- Procedure call statements that call user-defined procedures or NOS/VE procedures.
- Parameters of type pointer to procedure.
- ALLOCATE, FREE, PUSH, or NEXT statements that have parameters that are not local variables.

## Parameter List

A parameter list is an optional list of variable declarations that appears in the first statement of the function declaration. In the function declaration format shown earlier, they are shown as *formal_ parameters*. Declarations for formal parameters must appear in that first statement; they cannot appear in the declaration list in the body of the function.

A parameter list allows you to pass values from the calling program to the function. When a call is made to a function, parameters called actual parameters are included with the function name. The values of those actual parameters replace the formal parameters in the parameter list. Wherever the formal parameters exist in the statements within the function, the values of the corresponding actual parameters are substituted. For every formal parameter in a function declaration, there must be a corresponding actual parameter in the function call.

There are two kinds of parameters: reference parameters and value parameters. A reference parameter has the form:

> **VAR name** *{,name}...* **: type**
> *{,name {,name}... : type}...*

A value parameter has the form:

> **name** *{,name}...* **: type**
> *{,name {,name}... : type}...*

In procedures, reference parameters and value parameters cause different actions to be taken; in functions, however, both kinds of parameters have the same effect. (In a procedure, the value of a reference parameter can change during execution of the procedure; a value parameter cannot change.) In a function, neither reference parameters nor value parameters can change in value. A formal reference parameter can be any fixed or adaptable type. A formal value parameter can be any fixed or adaptable type, except a heap or an array or record that contains a heap.

Reference parameters and value parameters can be specified in many combinations. When both kinds of parameters appear together, they must be separated by semicolons. Parameters of the same type can also be separated by semicolons instead of commas, but in this case, VAR must appear with each reference parameter. All of the following parameter lists are valid.

- `VAR i, j: integer; a, b: char;`
- `VAR i: integer; VAR j: integer; a: char; b: char;`
- `a: char; VAR i, j: integer; b: char;`
- `VAR i: integer, j: real; a: char, b: boolean;`

In each of the preceding examples, I and J are reference parameters; A and B are value parameters.

## Referencing a Function

The call to the function is usually contained in an expression. The call consists of the function name (as given in the function declaration) and any parameters to be passed to the function in the following format:

**name** (*{actual_parameters}* )

**name**

Name of the function.

*actual_parameters*

Zero or more expressions or variables to be substituted for formal parameters defined in the function declaration. If you specify two or more, separate them with commas. They are substituted one-for-one based on their position within the list; that is, the first actual parameter replaces the first formal parameter, the second actual parameter replaces the second formal parameter, and so on. For every formal parameter in a function declaration, there must be a corresponding actual parameter in the function call.

If you did not specify any formal parameters in the function declaration, you can't include any actual parameters in the function call. However, you must enter left and right parentheses to indicate the absence of parameters. In this case, the call is:

**name( )**

The function can be anywhere that a variable of the same type could be. The value returned by a function is the last value assigned to it. If control is returned to the calling point before an assignment is made, results are unpredictable.

The only types that can be returned as values of functions are the basic types: scalar, floating point, pointer, and cell.

Example:

The following function finds the smaller of two integer values represented by formal value parameters A and B. The smaller value is assigned to MIN, the name of the function, and that integer value is returned.

```
FUNCTION min (a,
     b: integer): integer;
  IF a > b THEN
    min := b;
  ELSE
    min := a;
  IFEND;
FUNCEND min;
```

This function could be called using the following reference:

```
smaller := min (first, second);
```

The value of the variable FIRST is substituted for the formal parameter A; the value of SECOND is substituted for B. The value returned, the smaller value, replaces the entire function reference; the variable SMALLER is assigned the smaller value.

## Inline Functions

An inline function is one for which the compiler inserts the actual statements that are within the function at the point in the code where the function call was made. An inline function must be declared with the INLINE attribute. (Procedures can also be declared to be inline.)

Type, constant, and variable declarations that are local to an inline function or procedure are appended to the declarations of the function or procedure that called it. However, these types, constants, and variables can be referenced only within the body of the inline function or procedure; all the usual naming rules and scope rules still apply. (Local variable declarations in an inline function or procedure become part of the stack frame of the calling procedure.)

A variable declared within an inline function or procedure cannot be declared with the STATIC or XDCL attribute. Another function or procedure can be declared within an inline function or procedure only if it is declared with the XREF (externally declared) attribute; an inline function or procedure cannot contain any other function or procedure declarations.

Formal parameters in an inline function or procedure are treated as local variable declarations. When an inline function or procedure is called, the actual parameters are assigned to the corresponding formal parameters' local variables. Reference parameters are accessed by assigning a pointer to the actual parameter to the formal parameter's local variable.

When the actual parameter for a value parameter is an adaptable type or a substring, the parameter is treated as a read-only reference parameter (that is, a local copy of the parameter is not created). This is necessary to allow the type to be set at execution time. For adaptable array and adaptable record value parameters, the actual parameter must be byte-aligned.

An inline function or procedure can call any other function or procedure, including other inline functions and procedures, up to five levels. However, recursive calls to an inline function or procedure, either directly or indirectly, are not allowed. More than five nested calls and recursive calls are considered compile-time errors and end inline substitution.

Space that is allocated by a PUSH statement in an inline function or procedure is not released until the calling (not the inline) function or procedure completes.

The result of a reference to an inline function becomes part of the caller's stack frame. When an inline function is called more than once within a statement, the results of each reference are separate even though they share the same name.

The name of an inline function or procedure cannot be used in a pointer reference.

If a source listing is produced during compilation, the statements of the inline function or procedure are not listed at the point where the call occurs.

You can use the Debug Utility with inline functions and procedures. Debug treats an inline procedure call as a series of statements on the same line as the procedure call itself. It treats an inline function call as a series of statements on the same line as the end of the phrase that contained the reference to the inline function. Debug may not be able to access local variables declared in an inline function or procedure directly because the substitution process could create duplicate variable names (for example, if the names have already been used in the calling function or procedure). In that case, the Debug Utility always gives precedence to the variable names used in the calling procedure. For further information on the Debug Utility, refer to chapter 9.

## System-Dependent Functions

The functions described here can be used with CYBIL only on NOS/VE. As you review this section, keep in mind that programs using these functions cannot be transported to other operating systems and run on variations of CYBIL.

To use these functions properly and efficiently, you should be familiar with basic hardware concepts of your computer system. This information can be found in volume II of the virtual state hardware reference manual.

The functions are described in alphabetical order.

## #ADDRESS Function

The #ADDRESS function accepts a ring number, segment number, and byte offset and returns a value that is of type pointer to cell.

Use this format for the #ADDRESS function call:

**#ADDRESS(ring, segment, offset)**

**ring**

Ring number, ranging from 1 to 15.

**segment**

Segment number, ranging from 0 to 4,095.

**offset**

Byte offset, ranging from −80000000 hexadecimal to 7FFFFFFF hexadecimal.

Example:

The following example uses the #ADDRESS function to set the variable PTR1 to a pointer to cell formed using a ring number of 11, a segment number of 10, and a byte offset of 0FFFF hexadecimal.

```
VAR
  i,
  j,
  k: integer,
  ptr1: ^cell;

i := 11;
j := 10;
k := 0ffff(16);
ptr1 := #address (i, j, k);
```

## #FREE_RUNNING_CLOCK Function

The #FREE_RUNNING_CLOCK function returns the value of the free running microsecond clock.

Use this format for the #FREE_RUNNING_CLOCK function call:

**#FREE_RUNNING_CLOCK(port)**

**port**

An integer expression whose value is 0 or 1. It specifies the memory port to be used for reading the clock.

The integer value returned is that of the free running clock that is maintained within the memory connected to the specified processor memory port.

For further information on the free running microsecond clock and memory ports, refer to volume II of the virtual state hardware reference manual.

Example:

The following example sets the integer variable I to the value of the free running microsecond clock in the memory connected to processor memory port 0.

```
VAR
  i: integer;

i := #free_running_clock (0);
```

## #OFFSET Function

The #OFFSET function accepts a pointer and returns the integer value of the signed offset (byte number) contained in the pointer.

Use this format for the #OFFSET function call:

**#OFFSET(pointer)**

**pointer**

Name of a pointer expression.

A pointer consists in part of the process virtual address (PVA) of the first byte of the object to which it is pointing. An element of the PVA is the byte number. This byte number is the signed offset returned.

For further information on PVAs, refer to volume II of the virtual state hardware reference manual.

Example:

The following example finds the byte offset in the pointer PTR1.

```
VAR
  ptr1: ^cell,
  byte_offset: - 80000000(16) .. 7fffffff(16);
      :
byte_offset := $offset (ptr1);
```

If PTR1 was formed using the following #ADDRESS function,

```
ptr1 := #address (11, 10, 0ffff(16));
```

the value of BYTE_OFFSET would be 0FFFF hexadecimal.

## #PREVIOUS_SAVE_AREA Function

The #PREVIOUS_SAVE_AREA function returns a pointer to the first cell of the previous save area.

Use this format for the #PREVIOUS_SAVE_AREA function call:

**#PREVIOUS_SAVE_AREA** ( )

A procedure uses an area called a stack frame to store its automatic variables. If another procedure is called, hardware saves certain registers of the calling procedure and puts them in a stack frame save area. These registers contain the information required for the calling procedure to resume normal execution when control is returned by the called procedure.

If procedure calls are nested, each subsequent call creates its own stack frame save area and the last save area becomes the previous save area. Pointers are kept to link the previous save areas so that as procedures complete and return, the system works back through the previous save areas using the information contained in them to resume each procedure.

The formats of the stack frame save area and previous save area are shown in the CYBIL System Interface manual. For further information on the stack frame save area and previous save area, refer to volume II of the virtual state hardware reference manual.

Example:

The following example sets the pointer variable PSA_PTR to point to the first cell of the previous save area. The #CALLER_ID procedure then returns information about the caller of the last function. That information is returned in the record CALLER_RECORD. In this example, CALLER_RECORD is equivalent to the object of pointer PSA_PTR (that is, CALLER_RECORD = PSA_PTR^).

```
TYPE
  id_rec = record
    id: 0 .. 0ffffffff(16),
  recend;

VAR
  psa_ptr: ^id_rec,
  caller_record: id_rec;

psa_ptr := #previous_save_area ();
#caller_id (caller_record);
```

## #READ_REGISTER Function

The #READ_REGISTER function performs actions equivalent to the copy from state register (CPYSX) hardware instruction. It allows a program to read the contents of a process or processor register.

Use this format for the #READ_REGISTER function call:

**#READ_REGISTER(register_id)**

**register_id**

An integer expression from 0 to 255 that identifies the number of the register to be read. Register numbers are given in volume II of the virtual state hardware reference manual.

An integer value is returned.

The #WRITE_REGISTER procedure described in chapter 7 allows a program to change the contents of a process or processor register.

For further information on process and processor registers, and the CPYSX instruction, refer to volume II of the virtual state hardware reference manual.

Example:

The following example sets the integer variable J to the value of register E5, the Debug mask register.

```
VAR
  j: integer;

j := #read_register (0e5(16));
```

## #RING Function

The #RING function accepts a pointer and returns the integer value of the ring number contained in the pointer.

Use this format for the #RING function call:

**#RING(pointer)**

**pointer**

Name of a pointer expression.

Example:

The following example finds the ring number in the pointer PTR1.

```
VAR
  ptr1: ^cell,
  ring_number: integer;
      :
ring_number := #ring (ptr1);
```

If PTR1 was formed using the following #ADDRESS function,

```
ptr1 := #address (11, 10, 0ffff(16));
```

the value of RING_NUMBER would be 11.

## #SEGMENT Function

The #SEGMENT function accepts a pointer and returns the integer value of the segment number contained in the pointer.

Use this format for the #SEGMENT function call:

**#SEGMENT(pointer)**

**pointer**

Name of a pointer expression.

Example:

The following example finds the segment number in the pointer PTR1.

```
VAR
  ptr1: ^cell,
  segment_number: integer;
      :
segment_number := #segment (ptr1);
```

If PTR1 was formed using the following #ADDRESS function,

```
ptr1 := #address (11, 10, 0ffff(16));
```

the value of SEGMENT_NUMBER would be 10.

# Procedures 7

This chapter describes the procedures that are predefined in CYBIL and explains how you can define your own procedures.

# Procedures 7

A procedure is one or more statements that perform a specific action and can be called by a single statement. A procedure allows you to associate a name with the statement list so that by specifying the name itself as if it were a statement, you cause the list to be executed. Declarations can be included and take effect when the procedure is called. A procedure call can optionally cause actual parameters included in the call to be substituted for the formal parameters in the procedure declaration before the procedure's statements are executed.

A procedure differs from a function in that:

- A procedure can, but does not always, return a value.
- The call to a procedure is the procedure's name itself; a function call by contrast must be part of an expression in a statement.
- There can be no value assigned to the procedure name as there is to a function name.

Procedures can be recursive; that is, a procedure can call itself. In that case, the code within the procedure must not modify itself. Appendix F, The CYBIL Run-Time Environment, describes how recursive procedures are managed.

You can call standard procedures that are already defined in the CYBIL language, you can define your own procedures, or you can call procedures designed specifically for use on NOS/VE. This chapter describes all three.

# Standard Procedures

The STRINGREP procedure described here is a standard CYBIL procedure. It can be used safely in variations of CYBIL available on other operating systems. The last section in this chapter, System-Dependent Procedures, describes procedures that may not be available on other operating systems or that are unique to CYBIL on NOS/VE.

## STRINGREP Procedure

The STRINGREP procedure converts one or more elements to a string of characters, then returns that string and the length of the string.

Use this format for the STRINGREP procedure call:

**STRINGREP(string_name, length, element** *{,element}...***);**

**string_name**

Name of a string type variable. (You can specify it as a substring.) The result is returned here. It will contain the character representations of the named element(s).

**length**

Name of an integer variable. The procedure will set its value to the length in characters of the resulting string variable, string_name. It will be less than or equal to the declared length of the string variable.

**element**

Name of the element to be converted. The element can be a scalar, floating-point, pointer, or string type. Formats for specifying particular types and rules for conversion of those types are discussed in more detail later in this chapter.

The named elements are converted to strings of characters. Those strings are then concatenated and returned left-justified in the named string variable. The length of the string variable is also returned. If the result of concatenating the string representations is longer than the length of the string variable, the representation of the last element is a string of asterisk characters; the length that will be returned is the length of the string variable.

Each individual element is converted and placed in a temporary field before concatenation with other elements. The length of the temporary field can be specified as part of the element parameter that is described in the following sections. Generally, numeric values are written right-justified in the temporary field with spaces added on the left to fill the field, if necessary. String or character values are written left-justified in the temporary field with spaces added on the right to fill the field, if necessary. For both numeric and alphabetic values, the field is filled with asterisk characters if it is too short to hold the resulting value. The value of the field length, when specified, must be greater than or equal to zero; otherwise, an error occurs.

The following paragraphs describe how the STRINGREP procedure converts specific types and how they appear in the temporary fields.

## Integer Element

Use this format to specify an integer element:

**expression** *{ : length } { : #(radix) }*

**expression**

An integer expression to be converted.

*length*

A positive integer expression specifying the length of the temporary field. The length must be greater than or equal to 2. If it is omitted, the temporary field is the size required to hold the integer value and the leading sign character.

*radix*

Radix of expression. Possible values are 2, 8, 10, and 16. If it is omitted, 10 is assumed.

The value of the integer expression is converted into a string representation in the desired radix. The resulting string representation is right-justified in the temporary field. If the expression is positive, a space precedes the leftmost significant digit. If the integer expression is negative, a minus sign precedes the leftmost significant digit. The leading space or hyphen must be considered a part of the length. Thus, the length must be greater than or equal to 2 in order to hold the sign character and at least one digit.

If you specify a field length larger than necessary, spaces are added on the left to fill the field. If you specify a field length that is not long enough to contain all digits and the sign character, the field is filled with a string of asterisk characters. If you specify a field length less than or equal to zero, an error occurs.

### Character Element

Use this format to specify a character element:

**expression** *{ : length }*

**expression**

A character expression to be converted.

*length*

A positive integer expression specifying the length of the temporary field. If it is omitted, a length of 1 is assumed.

A single character is left-justified in the temporary field. If you specify a field length larger than necessary, spaces are added on the right to fill the field. Including a radix for a character element causes a compilation error.

### Boolean Element

Use this format to specify a boolean element:

**expression** *{ : length }*

**expression**

A boolean expression to be converted.

*length*

A positive integer expression specifying the length of the temporary field. If it is omitted, a length of 5 is assumed.

Either of the 5-character strings ' TRUE' or 'FALSE' is left-justified in the temporary field. If you specify a field length larger than necessary, spaces are added on the right to fill the field. If you specify a field length that is not long enough to contain all five characters, the temporary field is filled with asterisk characters. Likewise, if the expression you supply is neither TRUE nor FALSE, the temporary field is filled with asterisk characters. Including a radix for a boolean element causes a compilation error to occur.

## Ordinal Element

The integer value of an ordinal expression is handled the same way as an integer element. Refer to the discussion under Integer Element earlier in this chapter.

## Subrange Element

A subrange element is handled the same way as the element of which it is a subrange.

## Floating-Point Element

Use this format to specify a floating-point element:

**expression** *{ : length { : fraction } }*

**expression**

A real expression to be converted. If the value is INFINITE or INDEFINITE, an error occurs.

*length*

A positive integer expression specifying the length of the temporary field. If it is omitted, the temporary field is the size required to hold the real value and the necessary leading character.

*fraction*

Positive integer expression specifying the number of fractional digits to be included in a fixed-point format. Specify a value less than or equal to "length - 2". If it is omitted, conversion to floating-point format is assumed.

A floating-point expression can be converted into either a fixed-point format or a floating-point format depending on the fraction parameter. If it is included, the expression is converted to fixed-point format; if it is omitted, the expression is converted to floating-point format.

*Fixed-Point Format*

The form

**expression** *{: length{: fraction}}*

causes the specified expression to be converted to a string in fixed-point format. The string will have the specified length with the specified number of fractional digits to the right of the decimal place. The expression is rounded off so that the specified number of fractional digits are present. If no positive digit appears to the left of the decimal point, a 0 (zero) is inserted.

When figuring the length required to hold the expression, the compiler counts all digits to the left of the decimal point (it also counts 0 if it appears alone), the decimal point, and the specified number of fractional digits that appear to the right of the decimal point. If the expression is negative, an extra space is required for the minus sign. If you specify a field length larger than necessary, spaces are added on the left to fill the field. If you specify a field length that is not long enough to contain all digits, the sign character, and the decimal point, the field is filled with a string of asterisk characters.

Examples:

| Value of Expression E | Format of Element | Resulting String |
|---|---|---|
| 1.23456 | E:6:2 | `'  1.23'` |
| −1.23456 | E:6:3 | `'-1.235'` |
| 0 | E:5:2 | `' 0.00'` |

*Floating-Point Format*

The form

**expression** *{: length}*

causes the specified expression to be converted to a string in floating-point format.

The length of the temporary field is determined somewhat differently from the other elements. The system defines a maximum number of digits that can be contained in the mantissa of a real number and the number of digits that can be in the exponent.

When the compiler figures the number of digits that will be in the mantissa, it first determines the number of spaces that must be present in the string. It allows for the number of digits in the exponent and four additional spaces: one for the sign of the expression (a space if positive, – if negative), one for the decimal point in the mantissa, one for the exponent character (E), and one for the sign of the exponent (+ or –). The total number of required spaces is subtracted from the specified field length. The compiler then compares the result (field length minus required spaces) and the maximum number of digits allowed in the mantissa, and takes the smaller of the two. That number is used as the number of digits in the mantissa when the compiler rounds the floating-point expression.

If a field length larger than necessary is specified, spaces are added on the left to fill the field. If the fixed size of the exponent is larger than necessary, zeroes are added on the left to fill the field. If the number that results from the subtraction of required spaces from the field length is less than 1, the field is filled with a string of asterisk characters.

Examples:

| Value of Expression E | Format of Element | Resulting String |
|---|---|---|
| 123.456 | E:10 | `' 1.23E+002'` |
| –123.456 | E:11 | `'-1.235E+002'` |

## Pointer Element

Use this format to specify a pointer element:

**pointer** *{ : length } { : #(radix) }*

**pointer**

A pointer reference to be converted.

*length*

A positive integer expression specifying the length of the temporary field. If you omit the field length, the temporary field is the size required to contain the pointer value.

*radix*

Radix of the pointer value. Possible values are 2, 8, 10, and 16. For NOS/VE, the default radix is 16.

The value of the pointer expression is converted into a string representation in the specified radix. It is right-justified in the temporary field. If you specify a field length larger than necessary, spaces are added on the left to fill the field. If you specify a field length that is not long enough to contain all the digits, the field is filled with a string of asterisk characters.

## String Element

Use this format to specify a string element:

**expression** *{ : length }*

**expression**

A string variable, string constant, or substring to be converted.

*length*

A positive integer expression specifying the length of the temporary field. If it is omitted, the field is the size required to contain the string expression.

A string expression is left-justified in the temporary field. If you specify a field length larger than necessary, spaces are added on the right to fill the field. If you specify a field length that is shorter than the length of the string, the temporary field is filled with a string of asterisk characters.

## User-Defined Procedures

This section describes how you define your own procedures.

### Procedure Declaration

You define your own procedure with the procedure declaration.

Use this format to declare a procedure:

> **PROCEDURE** *{[attributes]}* **name** *{(formal_parameters)}*;[1]
> *{declaration_list;}*
> *{statement_list;}*
> **PROCEND** *{name}* ;

*attributes*

Specify one or more of the following attributes. If you specify more than one attribute, separate them with commas.

XREF

The procedure has been compiled in a different module. In this case, the procedure declaration can contain the name and formal parameters, but no declaration list or statement list. In the other module, the procedure must have been declared with the XDCL attribute and an identical parameter list. If it is omitted, the procedure must be defined within the module where it is called.

XDCL

The procedure can be called from outside the module in which it is located. This attribute can be included only in a procedure declared at the outermost level of a module; it cannot be contained in a program, function, or another procedure. Other modules that call this procedure must contain the same procedure declaration with the XREF attribute specified.

1. Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a procedure declaration. If it is included in a CYBIL program run on NOS/VE, this parameter is ignored.

INLINE

Instead of calling the procedure, the compiler inserts the actual procedure statements at the point in the code where the procedure call is made. Additional information on inline procedures is given later in this section.

#GATE[2]

The procedure can be called by a procedure at a higher ring level if the call is issued from within the call bracket of the gated procedure.[3] If you specify #GATE, you must also specify the XDCL attribute.

If you don't specify any attributes, the procedure is assumed to be in the same module in which it is called.

**name**

Name of the procedure. The procedure name is optional following PROCEND.

*formal_parameters*

One or more parameters in the form:

**VAR name** *{,name}...* **: type**
*{,name {,name}... : type}...*

and/or:

**name** *{,name}...* **: type**
*{,name {,name}... : type}...*

2. This attribute is not supported on variations of CYBIL available on other operating systems.

3. A ring level is a hardware feature. Rings provide hardware protection in that an unauthorized program cannot access anything at a lower ring level. For further information on rings, refer to the SCL Object Code Management manual.

The first form is called a reference parameter; its value can be changed during execution of the procedure. The second form is called a value parameter; its value cannot be changed by the procedure. Both kinds of parameters can appear in the formal parameter list; if so, separate them with semicolons (for example, I:INTEGER; VAR A:CHAR). Reference and value parameters are discussed in more detail later in this chapter under Parameter List.

The maximum number of parameters that can be passed to an externally referenced (XREF) procedure is 127.

*declaration_list*

Zero or more declarations.

*statement_list*

Zero or more statements.

The maximum number of user-defined procedures allowed in a single compilation unit is 999. Procedures can be nested up to 50 levels.

## Parameter List

A parameter list is an optional list of variable declarations that appears in the first statement of the procedure declaration. In the procedure declaration format shown earlier, they are shown as *formal_parameters*. Declarations for formal parameters must appear in that first statement; they cannot appear in the declaration list in the body of the procedure.

A parameter list allows you to pass values from the calling program to the procedure. When a call is made to a procedure, parameters called actual parameters are included with the procedure name. The values of those actual parameters replace the formal parameters in the parameter list. Wherever the formal parameters exist in the statements within the procedure, the values of the corresponding actual parameters are substituted. For every formal parameter in a procedure declaration, there must be a corresponding actual parameter in the procedure call.

There are two kinds of parameters: reference parameters and value parameters. A reference parameter has the form:

> **VAR name** *{,name}...* **: type**
> *{,name {,name}... : type}...*

When a reference parameter is used, the formal parameter represents the corresponding actual parameter throughout execution of the procedure. Thus, an assignment to a formal parameter changes the variable that was passed as the corresponding actual parameter. An actual parameter corresponding to a formal reference parameter must be addressable. A formal reference parameter can be any fixed or adaptable type. If the formal parameter is a fixed type, the actual parameter must be a variable or substring of an equivalent type. If the formal parameter is an adaptable type, the actual parameter must be a variable or substring whose type is potentially equivalent. (For further information on potentially equivalent types, refer to Equivalent Types in chapter 4.)

A value parameter has the form:

> **name** *{,name}...* **: type**
> *{,name {,name}... : type}...*

When a value parameter is used, the formal parameter takes on the value of the corresponding actual parameter. However, the procedure cannot change a value parameter by assigning a value to it or using it as an actual reference parameter to another procedure or function. A formal value parameter can be any fixed or adaptable type except a type that cannot have a value assigned, that is, a heap, or an array or record that contains a heap. If the formal parameter is a fixed type, the actual parameter can be any expression that could be assigned to a variable of that type. Strings must be of equal length. If the formal parameter is an adaptable type, the current type of the actual parameter must be one to which the formal parameter can adapt. If the formal parameter is an adaptable pointer, the actual parameter can be any pointer expression that could be assigned to the formal parameter. Both the value and the current type of the actual parameter are assigned to the formal parameter.

Reference parameters and value parameters can be specified in many combinations. When both kinds of parameters appear together, they must be separated by semicolons. Parameters of the same type can also be separated by semicolons instead of commas, but in this case, VAR must appear with each reference parameter. All of the following parameter lists are valid:

- `VAR i, j: integer; a, b: char;`
- `VAR i: integer; VAR j: integer; a: char; b: char;`
- `a: char; VAR i, j: integer; b: char;`
- `VAR i: integer, j: real; a: char, b: boolean;`

In each of the preceding examples, I and J are reference parameters; A and B are value parameters.

## Calling a Procedure

A call to a procedure consists of the procedure name (as given in the procedure declaration) and any parameters to be passed to the procedure in the following format:

**name** *{(actual_parameters)}* ;

**name**

Name of the procedure or a pointer to a procedure.

*actual_parameters*

One or more expressions or variables to be substituted for formal parameters defined in the procedure declaration. If you specify two or more, separate them with commas. They are substituted one-for-one based on their position within the list; that is, the first actual parameter replaces the first formal parameter, the second actual parameter replaces the second formal parameter, and so on. For every formal parameter in a procedure declaration, there must be a corresponding actual parameter in the procedure call.

A procedure is a type, like the types described in chapter 4. Procedure types are used for declaration of pointers to procedures; there are no procedure variables.

The lifetime of a formal parameter is the lifetime of the procedure in which it is a part. Storage space for the parameter is allocated when the procedure is entered and released when the procedure is left.

The lifetime of a variable that is allocated using the storage management statements (described in chapter 5) is the time between the allocation of storage (with the ALLOCATE statement) and the release of storage (with the FREE statement).

Two procedure types are equivalent if corresponding parameter segments have the same number of formal parameters, the same methods of passing parameters (reference or value), and equivalent types.

Example:

This example calculates the greatest common divisor X of M and N. M and N are passed as value parameters; that is, their values are used but M and N themselves are not changed. X, Y, and Z are reference parameters (preceded by the VAR keyword). Their original values are not used in this procedure; they are assigned new values in the procedure that destroy their previous values.

```
PROCEDURE gcd (m,
      n: integer;
  VAR x,
      y,
      z: integer);
{Extended Euclid's Algorithm}
  VAR
    a1,
    a2,
    b1,
    b2,
    c,
    d,
    q,
    r: integer;

  a1 := 0;
  a2 := 1;
  b1 := 1;
  b2 := 0;
  c := m;
  d := n;
```

```
    WHILE d <> 0 DO
    {a1 * m + b1 * n = d, a2 * m + b2 * n = c}
    {gcd (c,d) = gcd (m,n)}
      q := c DIV d;
      r := c MOD d;
      a2 := a2 - q * a1;
      b2 := b2 - q * b1;
      c := d;
      d := r;
      r := a1;
      a1 := a2;
      a2 := r;
      r := b1;
      b1 := b2;
      b2 := r;
    WHILEND;
    x := c;
    y := a2;
    z := b2;
    {x = gcd (m,n), y * m + z * n = gcd (m,n)}
  PROCEND gcd;
```

## Inline Procedures

An inline procedure is one for which the compiler inserts the actual statements that are within the procedure at the point in the code where the procedure call was made. An inline procedure must be declared with the INLINE attribute. (Functions can also be declared to be inline.)

Type, constant, and variable declarations that are local to an inline function or procedure are appended to the declarations of the function or procedure that called it. However, these types, constants, and variables can be referenced only within the body of the inline function or procedure; all the usual naming rules and scope rules still apply. (Local variable declarations in an inline function or procedure become part of the stack frame of the calling procedure.)

A variable declared within an inline function or procedure cannot be declared with the STATIC or XDCL attribute. Another function or procedure can be declared within an inline function or procedure only if it is declared with the XREF (externally declared) attribute; an inline function or procedure cannot contain any other function or procedure declarations.

Formal parameters in an inline function or procedure are treated as local variable declarations. When an inline function or procedure is called, the actual parameters are assigned to the corresponding formal parameter's local variables. Reference parameters are accessed by assigning a pointer to the actual parameter to the formal parameter's local variable.

When the actual parameter for a value parameter is an adaptable type or a substring, the parameter is treated as a read-only reference parameter (that is, a local copy of the parameter is not created). This is necessary to allow the type to be set at execution time. For adaptable array and adaptable record value parameters, the actual parameter must be byte-aligned.

An inline function or procedure can call any other function or procedure, including other inline functions and procedures, up to five levels. However, recursive calls to an inline function or procedure, either directly or indirectly, are not allowed. More than five nested calls and recursive calls are considered compile-time errors and end inline substitution.

Space that is allocated by a PUSH statement in an inline function or procedure is not released until the calling (not the inline) function or procedure completes.

The result of a reference to an inline function becomes part of the caller's stack frame. When an inline function is called more than once within a statement, the results of each reference are separate even though they share the same name.

The name of an inline function or procedure cannot be used in a pointer reference.

If a source listing is produced during compilation, the statements of the inline function or procedure are not listed at the point where the call occurs.

You can use the Debug Utility with inline functions and procedures. Debug treats an inline procedure call as a series of statements on the same line as the procedure call itself. It treats an inline function call as a series of statements on the same line as the end of the phrase that contained the reference to the inline function. Debug may not be able to access local variables declared in an inline function or procedure directly because the substitution process could create duplicate variable names (for example, if the names have already been used in the calling function or procedure). In that case, the Debug Utility always gives precedence to the variable names used in the calling procedure. For further information on the Debug Utility, refer to chapter 9.

## System-Dependent Procedures

Of the procedures described here, some can be used only with NOS/VE; others may be available in variations of CYBIL on other operating systems, but they are not guaranteed to be. Keep in mind that programs using these procedures may not be transportable to other systems.

To use these procedures properly and efficiently, you should be familiar with basic hardware concepts of your computer system. This information can be found in volume II of the virtual state hardware reference manual.

The functions are described in alphabetical order.

### #CALLER_ID Procedure

The #CALLER_ID procedure returns the identification (caller id) of the caller of a function or procedure. This procedure can be used only with NOS/VE.

Use this format for the #CALLER_ID procedure call:

**#CALLER_ID(id_record);**

**id_record**

Name of the record that will contain the caller id information. It must be four bytes long.

The caller id is a record that contains the key/lock fields, ring number, and segment number of the caller. (This information is found in the left half of the P register.) When a function or procedure is called, the caller id is placed in the leftmost 32 bits of the X0 register as a result of a call relative (CALLREL) or call indirect (CALLSEG) hardware instruction. The #CALLER_ID procedure accesses X0 while this information is there.

No special scope attributes (XDCL or XREF) are required in the calling function or procedure to use this procedure.

For further information on the caller id record and the CALLREL and CALLSEG instructions, refer to volume II of the virtual state hardware reference manual.

Example:

The following example sets the pointer variable PSA_PTR to point to the first cell of the previous save area. The #CALLER_ID procedure then returns information about the caller of the last function. That information is returned in the record CALLER_RECORD. In this example, CALLER_RECORD is equivalent to the object of pointer PSA_PTR (that is, CALLER_RECORD = PSA_PTR^).

```
TYPE
  id_rec = record
    id: 0 .. 0ffffffff(16),
  recend;

VAR
  psa_ptr: ^id_rec,
  caller_record: id_rec;

psa_ptr := #previous_save_area ();
#caller_id (caller_record);
```

## #COMPARE_SWAP Procedure

The #COMPARE_SWAP procedure performs actions equivalent to the compare swap (CMPXA) hardware instruction. It compares the contents of a variable with an expression. If the variable is unlocked and equal to the expression, the variable is swapped with a new expression. This procedure can be used only with NOS/VE.

Use this format for the #COMPARE_SWAP procedure call:

**#COMPARE_SWAP(lock_variable, initial_expression, new_expression, actual_variable, result_variable);**

**lock_variable**

Name of the variable on which the compare swap operation is to be performed. This variable must be aligned on a word boundary.

**initial_expression**

Expression that is compared to the lock variable. They must be equal for the swap operation to occur.

**new_expression**

Expression that specifies the value to be stored in the lock variable if the swap is successful (that is, the contents of lock_variable equals initial_expression). The expression must be greater than zero and less than $2^{32}-1$.

**actual_variable**

Name of the variable into which the initial contents of the lock variable is returned. If the lock variable is locked, this field is not changed.

**result_variable**

Name of the variable into which the result of the compare swap instruction is returned. Specify a subrange from 0 to 2 where each value has the following significance:

0

Swap operation was successful.

1

Swap operation failed because the initial expression was not equal to the contents of the lock variable.

2

Swap operation failed because the lock variable was locked.

The types of the lock variable, initial expression, new expression, and actual variable must be equivalent and have a size of eight bytes.

The lock variable is said to be locked if the leftmost 32 bits are ones. If it is locked, no action occurs. If it is unlocked, the contents of the lock variable is assigned to the actual variable. Then the lock variable is compared to an initial expression. If they are equal, a new expression is assigned to the lock variable. Otherwise, no swap occurs.

This procedure essentially performs the following statements:

```
IF (left half of lock_variable) = 0ffffffff(16) THEN
   result_variable := 2;
ELSE
   actual_variable := lock_variable;
   IF lock_variable = initial_expression THEN
      lock_variable := new_expression;
      result_variable := 0;
   ELSE
      result_variable := 1;
   IFEND;
IFEND;
```

These statements are executed by the hardware as a noninterruptable sequence. Access to the lock_variable from other sources, such as another processor or peripheral processor (PP), is prevented while these statements are being executed.

For further information on the CMPXA instruction, refer to volume II of the virtual state hardware reference manual.

Example:

The following example compares the variable LOCK with INITIAL. If LOCK is unlocked and equal to INITIAL, the value of LOCK is replaced by the value of variable NEW. In this example, LOCK is unlocked and equal to INITIAL. Therefore, following completion of the procedure, LOCK is equal to NEW which is 10. The variable RESULT is 0 indicating that the swap was successful.

```
VAR
  lock,
  initial,
  new,
  actual: integer,
  result: 0 .. 2;

lock := 5;
initial := 5;
new := 10;
#compare_swap (lock, initial, new, actual, result);
```

## #CONVERT_POINTER_TO_PROCEDURE Procedure

The #CONVERT_POINTER_TO_PROCEDURE procedure converts a variable of the type pointer to procedure that has no parameters to a variable of the type pointer to procedure that can have parameters. This procedure may not be available on variations of CYBIL that execute on other operating systems.

Use this format for the #CONVERT_POINTER_TO_PROCEDURE procedure call:

**#CONVERT_POINTER_TO_PROCEDURE(pointer_1, pointer_2);**

**pointer_1**

Name of a pointer to procedure variable with no parameters.

**pointer_2**

Name of a pointer to procedure variable with an arbitrary parameter list.

Example:

The following example converts the variable PTR_TO_PROC1, a pointer to a procedure that has no parameters, to the variable PTR_TO_PROC2, a pointer to a procedure that does have parameters.

```
VAR
  ptr_to_proc1: ^procedure,
  ptr_to_proc2: ^procedure (arg1: integer,
  arg2: real);

ptr_to_proc1 := ^proc1;
#convert_pointer_to_procedure (ptr_to_proc1, ptr_to_proc2);
```

## #HASH_SVA Procedure

The #HASH_SVA procedure performs actions equivalent to the load page table index (LPAGE) hardware instruction. This instruction searches the system page table (SPT) for a given system virtual address (SVA). This procedure can be used only with NOS/VE.

Use this format for the #HASH_SVA procedure call:

**#HASH_SVA(sva_variable, index, count, result_variable);**

**sva_variable**

Name of the variable that contains the SVA for which the instruction will search.

**index**

Name of an integer variable that will contain a word index into the SPT. If the SVA is found, this index points to the SPT entry for the SVA. If the SVA is not found, it points to the last entry searched.

**count**

Name of an integer variable that will contain the number of SPT entries searched.

**result_variable**

Name of a boolean variable that is set to TRUE if the SVA is found.

The procedure returns either an index within the table if the SVA is found, or an index of the last entry searched if the SVA is not found. It also returns the number of entries searched and a boolean value indicating whether the entry was found.

For further information on the SVA, addressing, and the LPAGE instruction, refer to volume II of the virtual state hardware reference manual.

## #KEYPOINT Procedure

The #KEYPOINT procedure generates an inline keypoint hardware instruction based on parameters supplied in the call. It allows performance monitoring of programs using keypoint instructions as trap interrupts. This procedure can be used only with NOS/VE.

Use this format for the #KEYPOINT procedure call:

**#KEYPOINT(class, data, identifier);**

**class**

A constant integer expession from 0 to 15 that specifies the keypoint class. This value is placed in the j field of the hardware instruction.

**data**

A constant or variable expression from 0 to 0FFFFFFFF hexadecimal that specifies optional data to be collected with the keypoint. If you specify the constant 0, a 0 is placed in the k field of the hardware instruction. If you don't specify 0, the value is placed in an X register and that register is placed in the k field of the hardware instruction.

**identifier**

A constant expression from 0 to 0FFFF hexadecimal that specifies a keypoint identifier. It is placed in the Q field of the hardware instruction.

For further information on the KEYPOINT instruction, refer to volume II of the virtual state hardware reference manual.

## #PURGE_BUFFER Procedure

The #PURGE_BUFFER procedure performs actions equivalent to the purge hardware instruction. It purges the contents of cache or the map buffer. This procedure can be used only with NOS/VE. However, not all computer systems that support NOS/VE have cache and map buffers. If executed on a model without cache or map buffers, no action occurs.

Use this format for the #PURGE_BUFFER procedure call:

**#PURGE_BUFFER(option_value, address);**

**option_value**

A constant integer expression from 0 to 15 that specifies one of the following purge options:

0

Purge all entries in cache that are included in the 512-byte block defined by the system virtual address (SVA) in Xj.

1

Purge all entries in cache that are included in the active segment identifier (ASID) defined by the SVA in Xj.

2

Purge all entries in cache.

3

Purge all entries in cache that are included in the 512-byte block defined by the process virtual address (PVA) in Xj.

4-7

Purge all entries in cache that are included in the segment number defined by the PVA in Xj.

8

Purge all entries in the map (page table map if entries are kept in separate maps) relating to the page table entry defined by the SVA in Xj.

9

Purge all entries in the map (page table map if entries are kept in separate maps) relating to the page table entries that are included in the segment defined by the SVA in Xj.

10 or A(16)

Purge all entries in the map (page table map if entries are kept in separate maps) relating to the page table entry defined by the PVA in Xj.

11 or B(16)

Purge all entries in the map (both the page table and segment map) relating to the segment table entry defined by the PVA in Xj, and to all page table entries included within that segment.

12–15 or C(16)–F(16)

Purge all entries in the map.

**address**

Name of a 6-byte variable that specifies the PVA or SVA of the data to be purged.

For further information on addressing, cache and map buffers, and the purge instruction, refer to volume II of the virtual state hardware reference manual.

Example:

The following example purges all entries in cache that are in the block defined by the PVA in pointer variable PTR1.

```
VAR
  i: integer,
  ptr1: ^cell;

ptr1 := ^i;
#purge_buffer (3, ptr1);
```

## #SCAN Procedure

The #SCAN procedure scans a string from left to right until one of a specified set of characters is found or the entire string has been searched. This procedure may not be available on variations of CYBIL that execute on other operating systems.

Use this format for the #SCAN procedure call:

**#SCAN(scan_variable, string, index, result_variable);**

**scan_variable**

Name of the variable that indicates the character values for which the string is scanned. The variable must be 256 bits long. Each bit of the variable represents the character in the corresponding position of the ASCII character set. If a bit is set, the corresponding character is one for which the procedure scans.

**string**

String or substring to be scanned.

**index**

Name of an integer variable. If a character is found during scanning, the index of that character is returned in this variable. The index of a character is that character's position in the string; for example, the index value of the first character is 1. If no matching values are found, the variable contains the string length plus one.

**result_variable**

Name of a boolean variable that is set to TRUE if the scan finds one of the selected characters.

The procedure looks for any one character from a set of characters specified in a 256-bit variable. Bits are set in the variable to correspond to the characters in the same positions in the ASCII character set collating sequence. A set bit indicates that the procedure scans the string for the corresponding character. The procedure stops if it finds one of the characters specified. It returns the position of the character that caused termination and the boolean variable that indicates whether a character was found.

Example:

The following example searches the string variable SOURCE_STRING for the asterisk character (*). First, the character to be searched for (the asterisk) must be specified in the array variable SELECT. To do this, all 256 elements of SELECT are set to 0. Then the $INTEGER function is used to determine the position of the asterisk character in the ASCII character set collating sequence. The value returned in I is 42 (because the asterisk is in the forty-second position in the collating sequence). The forty-second position in the array SELECT is then set to 1. Assuming SOURCE_STRING contains an asterisk as the fifty-fourth character of the string, the value returned in INDEX is 54 and the value returned in RESULT is TRUE.

```
VAR
  source_string: string (100),
  select: packed array [0 .. 255] of 0 .. 1,
  i,
  index: integer,
  result: boolean;

FOR i := 0 TO 255 DO
  select [i] := 0;
FOREND;
i := $INTEGER ('*');
select [i] := 1;
#scan (select, source_string, index, result);
```

## #SPOIL Procedure

The #SPOIL procedure causes the compiler to inhibit optimization of the specified variables the next time they are referenced and instead load their values from memory. Subsequent references to the variables are again subject to optimization. This procedure may not be available on variations of CYBIL that execute on other operating systems.

Use this format for the #SPOIL procedure call:

**#SPOIL(name** *{, name}...*);

**name**

Name of one or more variables for which optimization should be inhibited. You can specify a maximum of 127 variable names.

When the CYBIL compiler generates optimized object code, variables may be moved or otherwise treated differently than if the code was not optimized. For example, local variables are stored in registers and carried there throughout execution rather than always being referenced from memory. Specifying a variable in the #SPOIL procedure call causes its value to be loaded from memory (rather than taken from a register) the first time it occurs in code after the #SPOIL procedure call. Following that reference, however, the compiler resumes optimizing of the variable. Inhibiting code optimization is sometimes necessary to control asynchronous uses of CYBIL variables. (For further information on optimization, refer to the description of the CYBIL command's OPTIMIZATION_LEVEL parameter in chapter 8.)

The CYBIL compiler interprets this procedure as it would an external procedure and treats each actual parameter specified as if it were associated with a reference (VAR) formal parameter. However, the compiler does not generate any code when the procedure is referenced.

Example:

The following example inhibits code optimizing for a variable, HEADER^, to ensure that it contains an accurate status value before a channel is unlocked. HEADER^ is a record variable that contains status and request codes. The WHILE statement loop checks the STATUS field of HEADER^, waiting for a change to occur on a disk which is being updated asynchronously. After the change on the disk occurs, a user-defined procedure UNLOCK_CHANNEL is called and the specified channel is unlocked. If the #SPOIL procedure was not included in the WHILE statement list and the code was optimized, the code that loads HEADER^.STATUS would be moved out of the WHILE loop and, because the disk is updated asynchronously, the value of the STATUS field would never change.

```
IF header^.request_code IN disk_access_set THEN
  WHILE header^.status = dsc$dft_no_response DO
    #SPOIL (header^);
  WHILEND;
  unlock_channel (idle_channel);
IFEND;
```

## #TRANSLATE Procedure

The #TRANSLATE procedure translates each character in a source field according to a translation table, and transfers the result to a destination field. This procedure may not be available on variations of CYBIL that execute on other operating systems.

Use this format for the #TRANSLATE procedure call:

**#TRANSLATE(table, source, destination);**

**table**

Name of a string variable whose length is 256 characters. This variable defines the translation table.

**source**

String to be translated.

**destination**

Name of a string variable into which the translated string is transferred.

Translation of the string occurs from left to right with each source byte used as an index into the translation table. Translated bytes from the table are stored in the destination field.

If the length of the source field is less than the length of the destination field, translated spaces fill the destination field. If the source field is larger than the destination field, the rightmost characters of the source field are truncated.

Example:

The following example translates a string named SOURCE_STRING according to an externally referenced translation table named TRANS1_TABLE. The resulting string is placed in DEST_STRING.

```
VAR
  trans1_table: [XREF] string (256),
  source_string: string (100),
  dest_string: string (100);

source_string (1, 10) := 'ten chars.';
#translate (trans1_table, source_string, dest_string);
```

## #UNCHECKED_CONVERSION Procedure

The #UNCHECKED_CONVERSION procedure copies directly from a source field to a destination field. This procedure may not be available on variations of CYBIL that execute on other operating systems.

Use this format for the #UNCHECKED_CONVERSION procedure call:

**#UNCHECKED_CONVERSION(source, destination);**

**source**

Name of a variable from which the copy is made.

**destination**

Name of a variable to which the copy is made.

The source and destination fields must have the same length in bits. Neither the source nor the destination field can be a pointer or contain a pointer. If either the source or destination field is the object of a pointer reference (pointer^), the pointer cannot be a pointer to a procedure.

The destination field must satisfy the same restrictions as the target of an assignment statement. This means that the destination field cannot be:

- A read-only variable
- A formal value parameter of the procedure that calls the #UNCHECKED_CONVERSION procedure
- A bound variant record
- The tag field name of a bound variant record
- A heap
- An array or record that contains a heap

Example:

The following example copies the contents of a 5-character string named SOURCE to a 5-element array named DESTINATION. After the operation, the contents of both variables are identical.

```
VAR
  source: string (5),
  destination: packed array [1 .. 5] of char;

#unchecked_conversion (source, destination);
```

## #WRITE_REGISTER Procedure

The #WRITE_REGISTER procedure performs actions equivalent to the copy to state register (CPYXS) hardware instruction. It allows a program to change the contents of a process or processor register. This procedure can be used only with NOS/VE.

Use this format for the #WRITE_REGISTER procedure call:

**#WRITE_REGISTER(register_id, data);**

**register_id**

An integer expression from 0 to 255 that identifies the number of the register to be written. Register numbers are given in volume II of the virtual state hardware reference manual.

**data**

Integer expression that contains the data to be written to the register.

The #READ_REGISTER function described in chapter 6 allows a program to read the contents of a process or processor register.

Writing to certain registers requires special privileges. For further information on process and processor registers, and the CPYXS instruction, refer to volume II of the virtual state hardware reference manual.

Example:

The following example changes the contents of register E5, the Debug mask register, to 1F hexadecimal.

```
VAR
  i: integer;

i := 01f(16);
#write_register (0e5(16), i);
```

# Compiling and Formatting Source Code 8

This chapter describes how to compile and format CYBIL source code.

# Compiling and Formatting Source Code 8

This chapter describes how to compile and format CYBIL source code. You compile source code using the CYBIL command and compile-time declarations, statements, and directives that you insert at the appropriate place in the code. You format source code using the FORMAT_CYBIL_SOURCE command and formatting directives likewise inserted in the code. (These directives in the code are called text-embedded directives.)

The CYBIL command and the FORMAT_CYBIL_SOURCE command are standard system commands. They use the same syntax and language elements for parameters that are described in the SCL Language Definition manual.

## Compiling Source Code

The CYBIL command compiles one or more modules of CYBIL source code. Compilation statements and directives are used to construct the unit to be compiled and to control that process. If the CYBIL command and a directive specify conflicting options, the directive encountered most recently is used. The CYBIL command, statements, and directives are described later in this section.

The maximum number of lines allowed in a single compilation unit is 65,535 if no run-time checking is performed. If run-time checking is selected, the maximum number of lines allowed is 32,767. In one compilation unit, there can be up to 16,383 unique names and up to 999 user-defined procedures. At most, 2,000 error messages can be generated for any one module within a compilation unit.

For further information on program execution, refer to the SCL Object Code Management manual.

## CYBIL Command

**Purpose** Calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced.

**Format** **CYBIL**

*INPUT=file*
*LIST=file*
*BINARY=file*
*LIST_OPTIONS=list of keyword*
*DEBUG_AIDS=list of keyword*
*ERROR_LEVEL=keyword*
*OPTIMIZATION_LEVEL=keyword*
*PAD=integer*
*RUNTIME_CHECKS=list of keyword*
*STATUS=status variable*

**Parameters** *INPUT* or *I*

The file that contains the source text to be read. You can specify a file position as part of the file name. Source input ends when an end-of-partition or an end-of-information is encountered on the source input file. If it is omitted, $INPUT is assumed.

*LIST* or *L*

The file on which the compilation listing is to be written. You can specify a file position as part of the file name. If you specify $NULL, all compile-time output is discarded. If it is omitted, $LIST is assumed.

*BINARY* or *B* or *BINARY_OBJECT* or *BO*

The file on which object code is to be written. You can specify a file position as part of the file name. If you specify $NULL, the compiler performs a syntactic and semantic scan of the program but does not generate object code. If it is omitted, $LOCAL.LGO is assumed.

*LIST_OPTIONS* or *LO*

A combination of the following list options. If you specify NONE, no list options are selected. If it is omitted, option S (list the source input file) is assumed.

A

Produces an attribute list of source input block structure and relative stack. The attribute listing is produced following the source listing on the file specified by the LIST parameter or, if you omit the LIST parameter, on file $LIST.

F

Produces a full listing. In effect, this option selects options A, S, and R.

O

Lists compiler-generated object code. When selected, this listing includes an assembly-like listing of the generated object code. This option has no effect if the BINARY_OBJECT parameter is set to $NULL.

R

Produces a symbolic cross-reference listing showing the location of a program entity definition and its use within a program.

RA

Produces a symbolic cross-reference listing of all program entities whether referenced or not.

S

Lists the source input file.

X

Used in conjunction with the compile-time directive LISTEXT so that listings can be externally controlled using the CYBIL command. The LISTEXT toggle must be ON. For further information, refer to the SET, PUSH, POP, and RESET directives later in this chapter.

*DEBUG_AIDS* or *DA*

A combination of the following debug options. If it is omitted, NONE (no debug options) is assumed.

ALL

Selects debug options DS and DT.

DS

Compiles all debugging statements. A debugging statement is a statement in the source text that is ignored unless this option is specified. These statements are enclosed by the compile-time directives COMPILE and NOCOMPILE (described later in this chapter).

DT

Generates debug tables (that is, the symbol table and line table) as part of the object code. These tables are used by the Debug Utility.

NONE

No debug options are selected.

*ERROR_LEVEL* or *EL*

One of the following error list options. If it is omitted, W (list warning and fatal diagnostics) is assumed.

F

Lists fatal diagnostics. If it is selected, only fatal diagnostics are listed.

W

Lists warning (informative) diagnostics as well as fatal diagnostics.

*OTIMIZATION_LEVEL* or *OL* or *OPTIMIZATION* or *OPT*

One of the following optimization options. If it is omitted, LOW is assumed.

DEBUG

Object code is stylized to facilitate debugging. Stylized code contains a separate packet of instructions for each executable source statement; it carries no variable values across statement boundaries in registers, and it notifies Debug each time the beginning of a statement or procedure is reached.

LOW

Provides for keeping constant values in registers.

HIGH

Provides for keeping local variables in registers, passing parameters to local procedures in registers, and eliminating redundant memory references, common subexpressions, and jumps to jumps. When this option is selected, the RUNTIME_CHECKS parameter cannot be specified.

*PAD*

The number of no-op (no operation) instructions generated between instructions that perform operations. If it is omitted, zero is assumed; no-op instructions are not generated.

*RUNTIME_CHECKS* or *RC*

A combination of the following run-time checking options. This parameter cannot be specified when OPTIMIZATION_LEVEL=HIGH is also specified. If it is omitted, NONE (no run-time checks) is assumed.

ALL

Selects run-time checking options N, R, and S.

N

Produces compiler-generated code that checks for a NIL value when a reference is made to the object of a pointer.

NONE

No run-time checks are produced.

R

Produces compiler-generated code to check ranges. Range checking code is generated for assignment to integer subranges, ordinal subranges, and character variables. All CASE statements are checked to ensure that the selection expression corresponds to one of the variant values specified if no ELSE clause is provided. All references to substrings are verified. If you specify an offset (variable pointer) on a RESET statement, it is checked to ensure that it is valid for the specified sequence.

S

Produces compiler-generated code to test the subscripting of arrays.

*STATUS*

An optional SCL status variable in which the completion status of the command is returned. If it is specified, the compiler returns a status to this variable indicating whether any fatal errors were found during the compilation that was just completed. You can test this status variable and take special action if fatal compilation errors occurred. If it is omitted and the status returned from the compiler is abnormal, SCL terminates the current command sequence.

**Remarks** If the compiler command specifies an option that differs from a directive, the latest occurrence of either the command or the directive takes precedence.

**Examples** This command reads source code from a file named COMPILE, writes the compilation listing on file LIST, and writes the object code on file BIN1. The listing includes source code, compiler-generated object code, and a symbolic cross-reference listing.

```
cybil i=compile l=list b=bin1 lo=(o,r)
```

## Compilation Declarations and Statements

Many program elements defined in CYBIL have counterparts that can be used to control the compilation process. They include variable declarations, expressions, and the assignment and IF statements. The IF statement is used to specify certain areas of code to be compiled. The IF statement requires the use of expressions, which in turn require variables. Assignment statements are used to change the value of variables and, thus, expressions.

### Compile-Time Variables

Only boolean type variables can be declared.

Use this format to specify a boolean type compile-time variable:

> **? VAR name** *{,name}...* **: BOOLEAN := expression**
> *{, name {,name}... : BOOLEAN := expression}...* **?;**

**name**

Name of the compile-time variable. This name must be unique among all other names in the program.

**expression**

A compile-time expression that specifies the initial value of the variable.

A compile-time declaration must appear before any compile-time variables are used. The scope of such a variable extends from the point at which it is declared to the end of the module. Compile-time variables can be used only in compile-time expressions and compile-time assignment statements. The maximum number of compile-time variables that can be used in a compilation unit is 1,023.

### Compile-Time Expressions

Compile-time expressions are composed of operands and operators like CYBIL-defined expressions. An operand can be:

- Either of the constants TRUE or FALSE
- A compile-time variable
- Another compile-time expression

The operators are NOT, AND, OR, and XOR. Their order of evaluation from highest to lowest is:

- NOT
- AND
- OR and XOR

These operators have their usual meanings, as described under Operators in chapter 5.

### Compile-Time Assignment Statement

A compile-time assignment statement assigns a value to a compile-time variable.

Use this format for the compile-time assignment statement:

**? name := expression ?;**

**name**

Name of a compile-time variable.

**expression**

A compile-time expression.

## Compile-Time IF Statement

The compile-time IF statement compiles or skips a certain area of code depending on whether a given expression is true or false.

Use this format for the compile-time IF statement:

> **? IF expression THEN**
>     **code**
>     *{ ? ELSE*
>     *code }*
> **? IFEND**

**expression**

A boolean compile-time expression.

**code**

An area of CYBIL code or text.

When the expression is evaluated as true, the code following the reserved word THEN is compiled. When compilation of that code is completed, compilation continues with the first statement following IFEND. When the expression is false, compilation continues following the ELSE phrase, if it is included, or following IFEND.

The ELSE clause is optional. If it is included, the ELSE clause designates an area of code that is compiled when the preceding expression is false.

Example:

The following example shows the declaration of a compile-time variable named SMALL_SIZE that is initialized to the value TRUE. A line of CYBIL code declaring an array named TABLE is compiled. Then a compile-time IF statement checks the value of SMALL_SIZE. If it is TRUE, the line of CYBIL code that calls a procedure named BUBBLESORT is compiled in the program. If it is FALSE, the code that calls procedure QUICKSORT is compiled instead. Because SMALL_SIZE was initialized to TRUE, the call to BUBBLESORT is included in the compiled program.

```
?VAR
   small_size: boolean := TRUE?;

VAR
  table: array [1 .. 50] of integer;

?IF small_size = TRUE THEN
  bubblesort (table);
?ELSE
  quicksort (table);
?
IFEND
```

## Compile-Time Directives

Compile-time directives allow you to perform many activities during compilation. They can be grouped into five major categories:

- Toggle control (the SET, PUSH, POP, and RESET directives)
- Layout control (the LEFT, RIGHT, EJECT, SPACING, SKIP, NEWTITLE, TITLE, and OLDTITLE directives)
- Maintenance control (the COMPILE and NOCOMPILE directives)
- Object code comment control (the COMMENT directive)
- Object library control (the LIBRARY directive)

You can turn on or off various listing options and run-time options using the SET and PUSH directives. The SET directive specifies new settings that replace the current settings. The PUSH directive, on the other hand, causes the current settings to be saved before initiating the new settings. In that case, the POP directive can be used to restore the last settings that were saved by the PUSH directive. The RESET directive restores the original settings and discards any settings that were saved.

The LEFT and RIGHT directives specify the margins of the source text to be read. Any text to the left of the left margin or the right of the right margin is ignored. The remaining layout control directives format the listing that results from compilation. The EJECT directive advances the paper to the top of the next page. The SPACING directive specifies single, double, or triple spacing between lines of the listing. The SKIP directive skips a specified number of lines. The TITLE, NEWTITLE, and OLDTITLE directives indicate titles that are printed on every page of the listing.

The COMPILE and NOCOMPILE directives specify areas of code which should or should not be compiled.

The COMMENT directive inserts a comment in the object module that is generated during compilation.

The LIBRARY directive associates one or more object libraries with the object module so that the loader can satisfy external references from those libraries.

You can specify one or more directives with the format:

**?? directive** *{,directive}...* **??**

**directive**

One of the directives discussed in the remainder of this chapter.

Directives must be bounded by a pair of consecutive question marks. These delimiters are not shown in the following formats for individual directives, but they are required around one or more directives.

If a directive conflicts with an option specified on the CYBIL command, the most recent directive takes precedence.

## COMMENT Directive

The COMMENT directive causes the compiler to include the given character string in the commentary portion of the object module generated by the compilation process.

Use this format for the COMMENT directive:

**COMMENT := 'character_string'**

**character_string**

A character string of up to 40 characters that specifies a compile-time comment.

This directive allows you to include comments in object modules so that the comments appear in the load maps. Any number of comments can be included, but only the last comment encountered appears.

Example:

```
?? COMMENT := 'Copyright 1985 by Control Data Corporation' ??
```

## COMPILE Directive

The COMPILE directive causes compilation to occur, or to resume after the occurrence of a NOCOMPILE directive.

Use this format for the COMPILE directive:

**COMPILE**

If you don't use either the COMPILE or NOCOMPILE directive, the COMPILE directive is assumed; source code is compiled.

When the CYBIL command includes the DEBUG_AIDS parameter with DS specified, debugging statements enclosed by the NOCOMPILE and COMPILE directives are compiled.

## EJECT Directive

The EJECT directive causes the paper to be advanced to the top of the next page.

Use this format for the EJECT directive:

**EJECT**

## LEFT Directive

The LEFT directive specifies the column number of the left margin of the source text.

Use this format for the LEFT directive:

**LEFT := integer**

**integer**

An integer value that represents the column number of the left margin. The left margin must be greater than zero.

All source text left of the left margin is ignored. If you don't use the LEFT directive, the left margin is assumed to begin in column 1.

The LEFT directive can also be used as a formatting directive. If it is encountered after a FORMAT_CYBIL_SOURCE command is issued, the formatter starts all of its formatted output at the left margin specified on the LEFT directive.

The RIGHT directive, described later in this section, specifies the column number of the right margin.

Example:

This example sets the left margin at column 1 and the right margin at column 110.

```
?? LEFT := 1, RIGHT := 110 ??
```

## LIBRARY Directive

The LIBRARY directive allows you to specify an object library from which external references in the compilation unit can be satisfied.

Use this format for the LIBRARY directive:

**LIBRARY := library_name**

**library_name**

A string constant that specifies the name of the library. This string must be a valid NOS/VE file name (although the compiler does not check for its validity). In addition, the string cannot contain the CAT (concatenation) operation or the $CHAR function.

As a result of this directive, the compiler includes the specified library name in the library record of the object module that is produced during compilation.[1] This allows externally referenced declarations to be linked with the appropriate object library. Even if the same library name is found in more than one directive, the library is entered in the library record of the object module only once.

Example:

This example will cause the loader to search the object library MY_CYB_LIBRARY to satisfy external references in the compilation unit.

```
?? LIBRARY := 'MY_CYB_LIBRARY' ??
```

1. For further information about the library record and the format of the object module, refer to the SCL Object Code Management manual.

## NEWTITLE Directive

The NEWTITLE directive specifies a new, additional title to be used on a page while saving the current title.

Use this format for the NEWTITLE directive:

**NEWTITLE := 'character_string'**

**character_string**

A character string specifying the title to be used. A single quote mark is indicated by two consecutive quote marks enclosed by quote marks [that is, ""].

The current title is saved and the given character string becomes the current title. A standard page header is always the first title printed on a page, followed by user-defined titles in the order in which they were saved. This means that titles are saved and restored in a last in-first out order, but they are printed in a first in-first out order. There is always a single empty line between the standard page header and any user-defined titles. There is always at least one empty line between the last title and the text.

The maximum number of titles that can be specified is 10. Any attempts to add more titles is ignored.

Titling does not take effect until the top of the next printed page.

## NOCOMPILE Directive

The NOCOMPILE directive causes compilation to stop until the occurrence of a COMPILE directive or the end of the module.

Use this format for the NOCOMPILE directive:

**NOCOMPILE**

NOCOMPILE continues listing source code and text according to the listing toggles and layout directives, interpreting and obeying directives, but source code is not compiled until a COMPILE directive is encountered or a MODEND statement is encountered.

When the CYBIL command includes the DEBUG_AIDS parameter with DS specified, debugging statements enclosed by the NOCOMPILE and COMPILE directives are compiled.

## OLDTITLE Directive

The OLDTITLE directive restores the last user-defined title that was saved, making it the current title.

Use this format for the OLDTITLE directive:

**OLDTITLE**

If there is no saved title, no action occurs.

## POP Directive

The POP directive restores the last toggle settings that were saved by the PUSH directive.

Use this format for the POP directive:

**POP**

If no record was kept (such as when a SET directive is performed), the initial settings are restored.

Example:

This example shows a PUSH directive that temporarily turns off listing. The POP directive restores listing.

```
?? PUSH (LIST := OFF) ??
      :
?? POP ??
```

## PUSH Directive

The PUSH directive specifies the setting of one or more toggles like the SET directive, but before the settings are put into effect, a record of the current state of all toggles is saved for later use.

Use this format for the PUSH directive:

**PUSH (toggle_name := condition** *{,toggle_name := condition}...)*

**toggle_name**

Name of the toggle being set. Listing toggles are described in table 8-1. Run-time checking toggles are described in table 8-2. The names of toggles can be used freely outside of directives.

**condition**

ON or OFF. If a toggle is ON, the activity associated with it is performed during compilation; if it is OFF, the activity is not performed.

Settings in the PUSH list are performed in the same manner as a SET list. If the directive list contains more than one setting for a single toggle, the rightmost setting in the list is used.

The POP directive, described earlier in this chapter, restores the original toggle settings in a last in-first out manner (that is, the last record to be saved is the first to be restored). A maximum of 25 toggle control directives can be stacked in a compilation unit.

Example:

This example turns off listing temporarily, that is, until the POP directive is encountered.

```
?? PUSH (LIST := OFF) ??
       :
?? POP ??
```

Table 8-1 describes the listing toggles and gives their initial settings.

**Table 8-1. Listing Toggles**

| Toggle | Initial Value | Description |
|---|---|---|
| LIST | ON | Determines whether other listing toggles are read. When ON, a source listing is produced and the other listing toggles are used to control other aspects of listing. When OFF, no listing is produced; the other listing toggles are ignored. |
| LISTOBJ | OFF | Controls the listing of generated object code. When ON, object code is interspersed with source code following the corresponding source code line. |
| LISTCTS | OFF | Controls the listing of the listing toggle directives and layout directives. |
| LISTEXT | OFF | When ON, the listing of source statements is controlled by a parameter (LIST_OPTIONS=X) on the CYBIL compiler command. |
| LISTALL | Not applicable | This option represents all of the listing toggles. When ON, all other listing toggles are ON; when OFF, all other listing toggles are OFF. |

Table 8-2 describes the run-time checking toggles and gives their initial settings. These initial settings apply only if the corresponding options were selected on the RUNTIME_CHECKS parameter of the CYBIL command.

**Table 8-2. Run-Time Checking Toggles**

| Toggle | Initial Value | Description |
|---|---|---|
| CHKRNG | ON | Controls the generation of object code that performs range checking of scalar subrange assignments and CASE statement variables. |
| CHKSUB | ON | Controls the generation of object code that checks array subscripts (indexes) and substring selections to verify that they are valid. |
| CHKNIL | OFF | Controls the generation of object code that checks for a NIL value when a reference is made to the object of a pointer. |
| CHKALL | Not applicable | This option represents all run-time checking toggles. When ON, all other run-time checking toggles are ON; when OFF, all other run-time checking toggles are OFF. |

## RESET Directive

The RESET directive restores the initial toggle settings.

Use this format for the RESET directive:

**RESET**

When the RESET directive is performed, any record of previous settings is destroyed.

## RIGHT Directive

The RIGHT directive specifies the column number of the right margin of the source text.

Use this format for the RIGHT directive:

**RIGHT := integer**

**integer**

An integer value that represents the column number of the right margin. The right margin must be greater than or equal to the left margin plus 10, and less than or equal to 110; that is:

left margin + 10 ≤ right margin ≤ 110

All source text right of the right margin is ignored. If nonblank characters appear in the source text after the right margin, the compiler places a vertical line in the source listing immediately following the right margin. This indicates that the compiler stopped scanning the line at that point.

If you don't use the RIGHT directive, the right margin is assumed to be column 79.

The RIGHT directive can also be used as a formatting directive. If it is encountered after a FORMAT_CYBIL_SOURCE command is issued, the formatter ends its formatted output at the right margin specified on the RIGHT directive.

The LEFT directive, described earlier in this section, specifies the column number of the left margin.

Example:

This example sets the left margin at column 1 and the right margin at column 110.

```
?? LEFT := 1, RIGHT := 110 ??
```

## SET Directive

The SET directive specifies the setting of one or more toggles.

Use this format for the SET directive:

**SET (toggle_name := condition** *{,toggle_name := condition}...)*

**toggle_name**

Name of the toggle being set. Listing toggles are described in table 8-1. Run-time checking toggles are described in table 8-2. The names of toggles can be used freely outside of directives.

**condition**

ON or OFF. If a toggle is ON, the activity associated with it is performed during compilation; if it is OFF, the activity is not performed.

All settings specified in the SET directive are done at the same time. If the directive list contains more than one setting for a single toggle, the rightmost setting in the list is used.

## SKIP Directive

The SKIP directive specifies that a given number of lines is to be skipped.

Use this format for the SKIP directive:

**SKIP := lines**

**lines**

Integer value specifying the number of lines to skip. Specify a value greater than or equal to 1.

If you specify more lines than the number of lines on the page, or if you specify a value for lines that would cause the paper to skip past the bottom of the current page, the paper is advanced to the top of the next page.

## SPACING Directive

The SPACING directive specifies the number of blank lines between individual lines of the listing.

Use this format for the SPACING directive:

**SPACING := spacing**

**spacing**

One of the values 1, 2, or 3 specifying single, double, and triple spacing, respectively.

An undefined value has no effect on spacing, but an error message is issued.

If you don't use the SPACING directive, single spacing (no intervening blank lines) is assumed.

## TITLE Directive

The TITLE directive replaces the current user-defined title with the given character string.

Use this format for the TITLE directive:

**TITLE := 'character_string'**

**character_string**

A character string specifying the title to be used. A single quote mark is indicated by two consecutive quote marks enclosed by quote marks [that is, ''''].

If there is no user-defined title currently, the character string becomes the current title.

A standard page header is always the first title printed on a page. There is always a single empty line between the standard page header and any user-defined titles. There is always at least one empty line between the last title and the text.

Titling does not take effect until the top of the next printed page.

## Formatting Source Code

Formatting CYBIL source code is useful because it improves the code's consistency, readability, and maintainability. The CYBIL source code formatter arranges source code using its own rules and, optionally, rules that you set for it. You can specify these optional formatting rules either as parameters on the FORMAT_CYBIL_SOURCE command or as directives embedded in the code itself.

With the FORMAT_CYBIL_SOURCE command, you can choose:

- Whether comments are set off by blank lines
- Whether the statements EXIT, CYCLE, and RETURN are specially marked to indicate a change in the flow of the program
- Whether successive spaces are compressed or left as is
- The line width of the formatted output line
- The key character that indicates Source Code Utility directives which should not be formatted

With the text-embedded directives FMT, LEFT, and RIGHT, you can choose:

- Whether or not the lines that follow the directive should be formatted or left as is
- Whether successive spaces are compressed or left as is
- Tab settings for specific characters
- The number of spaces to indent when indentation is called for
- The left and right margins of the formatted output (in effect, the initial left margin and the line width)

The option to compress space characters and the line width of the output line can be specified on both the command and the directives. If conflicting options are found, the most recent directive always takes precedence over the command.

Formatting directives are processed only after the FORMAT_CYBIL_SOURCE command is issued; otherwise, they are ignored. (The LEFT and RIGHT directives, however, can also be used as compilation directives. Refer to the individual descriptions of these directives for further information.) The command and directives are described later in this section.

The source code to be formatted does not have to be a complete compilation unit, but it should be syntactically correct. Each line can be a maximum of 256 characters. Multiple partitions on the source file are formatted.

Error messages are written to an error file. You can specify this file with the ERROR parameter on the FORMAT_CYBIL_SOURCE command. If you omit it, errors are written on the local file $ERRORS. An error does not cause formatting to stop; the entire source file is always processed.

Example:

The following example shows CYBIL source code as a user may have entered it and how it would look after formatting.

```
/copy_file $user.unformatted_program

procedure exit_example;
var i:integer, key:string(7),
names:[read] array [1..4] of string(7):=['jqp8402','jxd1432',
'efd3204','led4411'];
key:='efd3204';
/find_key/
for i:=lowerbound(names) to upperbound(names) do
if key=names[i] then exit /find_key/;
ifend;
forend/find_key/;
procend exit_example;
/format_cybil_source i=$user.unformatted_program ..
../o=$user.formatted_program
/copy_file $user.formatted_program

  PROCEDURE exit_example;

    VAR
      i: integer,
      key: string (7),
      names: [READ] array [1 .. 4] of string (7) :=
            ['jqp8402','jxd1432','efd3204','led4411'];

    key := 'efd3204';

  /find_key/
    FOR i := LOWERBOUND (names) TO UPPERBOUND (names) DO
      IF key = names [i] THEN
        EXIT /find_key/;
      IFEND;
    FOREND/find_key/;
  PROCEND exit_example;
/
```

## FORMAT_CYBIL_SOURCE Command

**Purpose** Formats CYBIL source code for consistency and greater readability.

**Format** **FORMAT_CYBIL_SOURCE** or
**FORCS**
*INPUT=file*
*OUTPUT=file*
*ERROR=file*
*FORMAT_OPTIONS=list of keyword*
*LINE_WIDTH=integer*
*KEY=character*
*STATUS=status variable*

**Parameters** *INPUT* or *I*

The file from which the CYBIL source code is read. The file path specified for the input file cannot be the same as the file path specified for the output file. If you don't specify a file position in the file reference, this file is rewound before formatting. If it is omitted, $INPUT is assumed.

*OUTPUT* or *O*

The file on which the formatted CYBIL source code is written. The file path specified for the output file cannot be the same as the file path specified for the input file. If you don't specify a file position in the file reference, this file is rewound before being written. If it is omitted, $OUTPUT is assumed.

*ERROR* or *E*

The file on which error messages are written. If it is omitted, $ERRORS is assumed.

*FORMAT_OPTIONS* or *FO*

One or more of the following format options. If it is omitted, NONE (no format options are selected) is assumed.

| | |
|---|---|
| ALL | Selects all the format options (CB, ME, and NC). |
| COMMENT_ BLOCK or CB | Specifies that a comment block is preceded and followed by a blank line. A comment block is considered to be one or more lines of comments. If a blank line already exists, none is added. Also, no blank lines are inserted within a comment block. |
| MARK_ EXIT or ME | Marks exit statements (that is, EXIT, CYCLE, and RETURN) by adding the comment `{----->` after the statement. This indicates that a change in the program's flow of control occurs here. If the comment delimiter (the left brace) is already in the statement, the statement is not changed. |
| NO_ COMPRESS or NC | Selects no compression of successive space characters. The same number of space characters in the input file are written to the output file. This option is overridden if the formatter finds an FMT directive in the source code with the COMPRESS parameter set to ON. Unless otherwise specified, spaces are compressed. |
| NONE | No format options are selected. |

*LINE_WIDTH* or *LW*

The line width of the formatted output. You can specify an integer from 11 to 110. Specifying a line width sets the left margin to column 1 and the right margin to the value of the line width. If it is omitted, a right margin of 79 is assumed.

This setting is overridden if the formatter finds the LEFT and RIGHT layout control directives in the source code.

*KEY* or *K*

The key character that indicates embedded Source Code Utility directives. Statements that begin with the key character in column 1 are not formatted. If it is omitted, the asterisk character is assumed.

*STATUS*

An optional SCL status variable in which the completion status of the command is returned.

**Remarks**

The file paths specified for the input and output files cannot be the same. For example, INPUT=PROC1 and OUTPUT=PROC1 are not valid; INPUT=$LOCAL.PROC1 and OUTPUT=$USER.PROC1, however, are valid.

**Examples**

This command formats the CYBIL source program contained on file INITIAL and writes it to file $USER.FINAL.

```
format_cybil_source initial $user.final
```

## Formatting Directives

You can insert directives in the source code itself that direct how formatting is done. These directives are the formatting directive FMT and the layout control directives LEFT and RIGHT. The FMT directive determines what formatting is done. The layout control directives set the margins for the formatted output.

You can specify one or more directives with the following format:

**?? directive** *{, directive}...* **??**

**directive**

One of the formatting directives FMT, LEFT, or RIGHT as described in the remainder of this section.

If an option specified on a directive differs from one selected on the FORMAT_CYBIL_SOURCE command, the most recent directive takes precedence.

## FMT Directive

The FMT directive controls formatting options such as when to format lines, whether multiple spaces are compressed, where tabs should be set for certain characters, and how to indent block-type statements. The options specified on the directive take effect starting with the next source code statement that is processed. They remain in effect until the entire file is formatted or another directive is encountered.

Use this format for the FMT directive:

> **FMT** (*FORMAT := keyword,*
> *COMPRESS := keyword,*
> *TAB := 'tab_character', integer {, integer}...,*
> *CLEARTAB := 'tab_character', integer {, integer}...,*
> *INDENT := integer*)

*FORMAT*

Specifies whether formatting is to take place. ON indicates that all source code lines, beginning with the line following this directive, are formatted. OFF indicates that all lines, beginning with the line following this directive, are not formatted. If it is omitted, ON (lines are formatted) is assumed.

*COMPRESS*

Specifies whether successive space characters are compressed to a single space. ON indicates that multiple space characters appearing together in the original code are compressed to one space character in the formatted code. OFF indicates that successive space characters are transferred to the formatted file unchanged. If it is omitted, ON (successive spaces are compressed) is assumed.

*TAB*

Tab settings that are used for the indicated characters. Use the following format to specify this parameter:

*'tab_character', integer {, integer }...*

The tab_character is from 1 to 8 characters. When the formatter recognizes that character or set of characters, it moves to the next available tab column as specified by the integers. For example, if the following TAB parameter was specified

```
?? fmt (tab:= '{', 5, 20) ??
```

and the formatter encountered a left brace in the first four columns of the source code, it would move the brace and the text following it to start at the fifth column. If the brace was found past the fifth column of the source code, it and the following text would be moved to start at the next tab position, column 20. If the specified character or characters are found past the last tab position that was specified, no text is moved; the characters stay in the same position.

The tab_character must be a valid CYBIL symbol (for example, { or :=). If a tab_character is specified, at least one integer setting must also be specified; no settings are assumed. The tabbing specified takes effect with the next line processed, not the line containing the TAB directive.

If it is omitted, no tab settings are used.

*CLEARTAB*

Tab settings that are deleted for the indicated tab_character. Use the following format to specify this parameter:

*'tab_character', integer {, integer }...*

The tab_character is from 1 to 8 characters. This directive deletes the tab settings specified for the indicated tab_character. For example, if the following CLEARTAB parameter was specified following the sample TAB parameter shown earlier,

```
?? fmt (cleartab:= '{', 5) ??
```

the formatter would no longer move to the fifth column when it encountered a left brace. Instead it would use the remaining tab setting (as set by the TAB parameter described earlier) and move to the twentieth column.

ALL can be used in place of the integers to indicate that all tab settings should be deleted for the specified tab character.

The tabbing specified takes effect with the next line processed, not the line containing the CLEARTAB directive.

*INDENT*

Number of columns to indent when a block-type statement is found. (These statements are BEGIN, FOR, REPEAT, WHILE, IF, and CASE; they are described in chapter 5.) This value is specified as an integer from 0 to 20. If it is omitted, two spaces is assumed. This indentation value does not apply to lines that are continued; continued lines are indented six spaces.

Example:

The following FMT directive causes formatting to be done without compressing successive space characters and setting tab positions at columns 1, 10, and 40 for the left brace character:

```
?? fmt (format:=on, compress:= off, tab:='{',1, 10, 40) ??
```

## LEFT Directive

The LEFT directive specifies the left margin used for the formatted output. In effect, this directive determines the base left margin from which all positioning and indenting take place. The source being used from the input file is not affected by this directive; only the output is affected.

Use this format for the LEFT directive:

**LEFT := integer**

**integer**

The column number of the left margin. The left margin must be greater than zero.

If you don't use the LEFT directive, the left margin of the formatted output is assumed to be column 1.

The LEFT directive can also be used during compilation (that is, following the CYBIL command) to indicate that all source text left of the left margin is ignored.

Example:

The following example sets the left margin of the formatted output at column 10 and the right margin at column 100. This means that the positioning and indenting of lines is based on an initial left margin of 10; for example, an indentation of 2 spaces moves the line to column 12. The maximum line width allowed in the formatted output using this example would be 90 characters.

```
?? left := 10 right := 99 ??
```

## RIGHT Directive

The RIGHT directive specifies the right margin used for the formatted output. In effect, this directive determines the line width used by the formatter. The source being used from the input file is not affected by this directive; only the output is affected.

Use this format for the RIGHT directive:

**RIGHT := integer**

**integer**

The column number of the right margin. The right margin must be greater than or equal to the left margin plus 10, and less than or equal to 110, that is:

left margin + 10 $\leq$ right margin $\leq$ 100

The line width of the formatted output can also be set on the FORMAT_CYBIL_SOURCE command. If so, the RIGHT directive overrides that value when it is encountered. If neither the FORMAT_CYBIL_SOURCE command nor the RIGHT directive specify line width, the right margin is assumed to be column 79.

The RIGHT directive can also be used during compilation (that is, following the CYBIL command) to indicate that all source text right of the right margin is ignored.

Example:

The following example sets the left margin of the formatted output at column 10 and the right margin at column 100. This means that the positioning and indenting of lines is based on an initial left margin of 10; for example, an indentation of 2 spaces moves the line to column 12. The maximum line width allowed in the formatted output using this example would be 90 characters.

```
?? left := 10 right := 99 ??
```

# Using the Debug Utility 9

# Using the Debug Utility 9

## Introduction to Debug

Debug is an SCL command utility that lets you debug a program during execution. Using Debug, you can stop execution at selected points, display the values of selected variables, and resume execution.

Debug requires no modification of your source code and no knowledge of assembly language. You can reference variables by their symbolic names rather than their addresses in memory. Furthermore, you do not need to interpret memory dumps or use a load map.

Debug can be used in line mode or screen mode. You can use Debug to perform machine-level debugging as well as symbolic debugging. This discussion focuses on using screen mode Debug for symbolic debugging. For information about line mode Debug, machine-level debugging, and other Debug features, see the Debug Usage manual.

Screen mode Debug gives you all of the Debug features with the ease of a full screen interface. You can execute Debug functions by pressing function keys rather than typing commands. Online HELP enables you to learn screen mode Debug as you use it.

Using the Debug utility in screen mode, you can:

- View your source code as it executes (an arrow points to the next line to be executed).
- Change the values of program variables while execution is suspended.
- Change the location where execution of your program resumes.
- View module components of your program.

## Getting Started

Using Debug in screen mode requires that your terminal support full screen operation. If your terminal is not set up for full screen operation, see the SCL System Interface manual for terminal definitions that support the full screen interface.

To execute your CYBIL program with Debug and use the symbolic debugging capability, you must compile the program with the OPTIMIZATION_LEVEL (OL) and DEBUG_AIDS (DA) parameters specified. Furthermore, to use Debug in screen mode, you must enter the command:

```
CHANGE_INTERACTION_STYLE STYLE=SCREEN
```

For example, to prepare the source program TEST_CYB contained in permanent file $USER.TEST_CYB for use with Debug, enter the following commands:

```
/change_interaction_style style=screen
/cybil input=$user.test_cyb binary=lgo ..
../optimization_level=debug debug_aids=all
```

or abbreviated,

```
/chais s=s
/cybil i=$user.test_cyb b=lgo ol=debug da=all
```

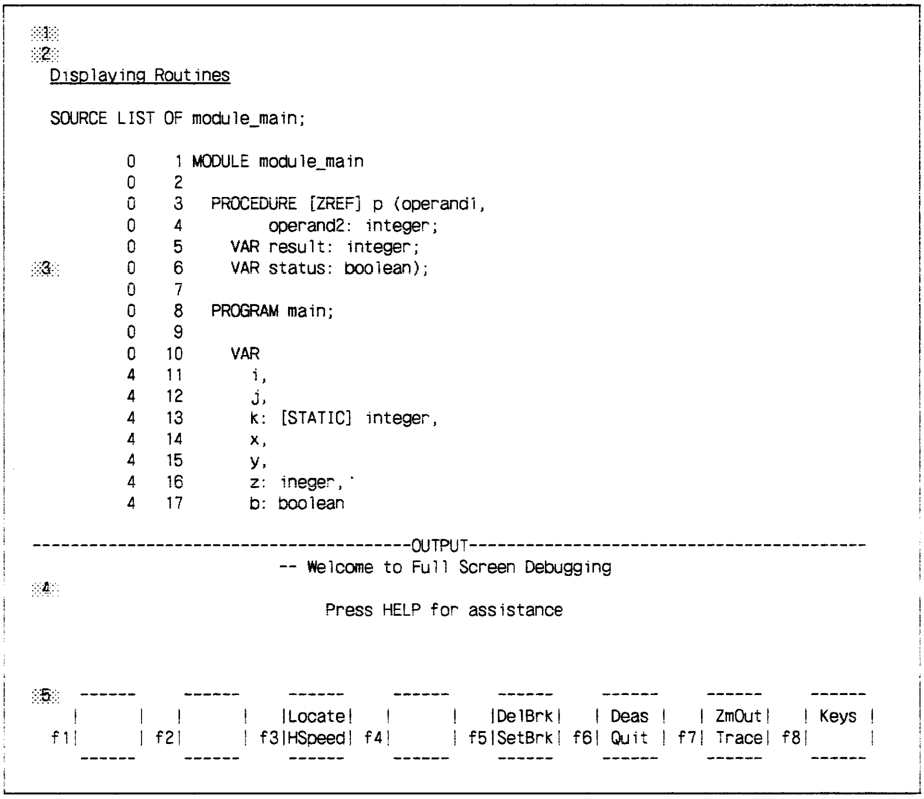
To execute TEST_CYB with screen mode Debug, enter the following command:

```
/execute_task file=lgo debug_mode=on
```

or abbreviated,

```
/exet f=lgo dm=on
```

On a Zenith Z19 or Heathkit H19 terminal the TEST_CYB source module is displayed as follows. (On other terminals, the screen format may vary slightly.)



```
 1
 2
   Displaying Routines

   SOURCE LIST OF module_main;

              0    1 MODULE module_main
              0    2
              0    3   PROCEDURE [ZREF] p (operandi,
              0    4         operand2: integer;
              0    5     VAR result: integer;
 3            0    6     VAR status: boolean);
              0    7
              0    8   PROGRAM main;
              0    9
              0   10     VAR
              4   11       i,
              4   12       j,
              4   13       k: [STATIC] integer,
              4   14       x,
              4   15       y,
              4   16       z: ineger,
              4   17       b: boolean

 ---------------------------------------OUTPUT-----------------------------------------
                          -- Welcome to Full Screen Debugging
 4
                               Press HELP for assistance


 5   ------     ------     ------     ------     ------     ------     ------     ------
   |      |  |      |  |Locate|  |      |  |DelBrk|  | Deas |  | ZmOut|  | Keys |
 f1|      | f2|      | f3|HSpeed| f4|      | f5|SetBrk| f6| Quit | f7| Trace| f8|      |
     ------     ------     ------     ------     ------     ------     ------     ------
```

**Figure 9-1. Debug Screen**

| | | |
|---|---|---|
| 1 | Home line | The line on which you enter Debug commands and SCL commands. |
| 2 | Response line | The line on which short responses and advisory messages from Debug are displayed. |
| 3 | Source window | The area in which the program you are debugging is displayed. |
| 4 | Output window | The area in which the output generated by your program (or output delivered by Debug) is displayed. |
| 5 | Row of function key assignments | The Debug functions assigned to function keys. Also, you can enter Debug commands on the home line. |

## How to Get Help

There are two ways to get help information while using the Debug utility in screen mode:

1. The HELP key.

   Pressing the HELP key displays the help window. The help window overlays a portion of your screen and prompts you to enter the item for which you need help. If you press a function key, a short description of the function you select is displayed in the help window. To exit HELP, press RETURN. Upon exiting HELP, your screen is restored to its original contents.

2. The EXPLAIN command.

   You can request help by entering the explain command on the HOME line. This command is used to read an online manual while you are debugging your program. To leave the online manual, press QUIT. When you leave the online manual, the screen is restored to its contents before you called EXPLAIN. For example, if you need information about Debug capabilities, press the HOME key and type the following EXPLAIN command on the HOME line:

   ```
   explain s='capabilities' m=debug
   ```

   This command takes you to the Debug online manual for an explanation of Debug capabilities. To return to screen mode Debug, press QUIT. See the SCL System Interface manual for more information about EXPLAIN.

## Example

This example demonstrates some commonly used Debug functions. It is represented as a series of steps. To get the most benefit from this example, you should create the sample program, EXAMPLE_CYB, illustrated in figure 9-2 then perform each step.

EXAMPLE_CYB is divided into the following test cases:

| | |
|---|---|
| TEST1 | A loop that increments a counter and then calls a procedure to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1 and STEPN functions. |
| TEST2 | A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table, and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data. |
| TEST3 | A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors. |

In each test case, the application of some Debug functions is demonstrated. After you work this example, you can begin to debug your CYBIL programs using screen mode Debug.

```
MODULE example_cyb;

{  Copy I/O procedures. }

      TYPE

         column = array [1..3] of string(3),
         twodim_array = array [1..6] of column;

      CONST

         maximum_record_length = 40;

      VAR

{  Declare program variables. }

         divisor     : real         := 0.0,
         dividend    : real         := 100.0,
         quotient    : real,
         cntr        : integer,
         result      : integer,
         month       : twodim_array,
         month_list  : string (18) := 'JANFEBMARAPRMAYJUN',
         month_row   : integer      := 0,
         length      : integer      := 10,
         i           : integer,

{  Declare variables for I/O. }

         lfn         : amt$local_file_name,
         o           : amt$file_identifier,
         s           : ost$status,
         f           : amt$file_byte_address,
         newline     : string (90),
         m1          : string (7)  := ' times ',
         m2          : string (3)  := ' = ',
         m3          : string (16) := ' The month is:  ',
         m4          : string (19) := ' The quotient is:  ';
```

**Figure 9-2. Example of an EXAMPLE_CYB Source Listing** *(Continued)*

*(Continued)*

```
   PROGRAM main;

{  These calls specify file attributes and open files. }

        lfn := '$OUTPUT';
        amp$open (lfn, amc$record, NIL, o, s);


{  TEST1:  Add to counter and call procedure SQUARE to square }
{          and display count.                                 }

        FOR cntr := 1 TO 10 DO
             square (cntr,result);
             stringrep(newline,length,' ',cntr:3,m1,cntr:3,
               m2,result:4);
             amp$put_next(o,^newline,maximum_record_length,
               f,s);
        FOREND:


{  TEST2:  Create single column table for each month. }

        WHILE month_row < 6 DO

             FOR i := 1 TO 3 DO
                month[month_row][i] := month_list
                  (month_row*3+1,3);
             FOREND;

        stringrep(newline,length,' ',m3,month[month_row]
          [i]:8);
        amp$put_next(o,^newline,maximum_record_length,f,s);
        month_row := month_row + 1;

        WHILEND;
```

**Figure 9-2. Example of an EXAMPLE_CYB Source Listing** *(Continued)*

*(Continued)*

```
{  TEST3:  Create divide fault. }

          quotient := dividend / divisor;
          stringrep(newline,length,' ',m4:19,quotient:6:1);
          amp$put_next(o,^newline,maximum_record_length,f,s);

      PROCEND main;

{  Procedure for squaring numbers. }

      PROCEDURE [XDCL] square (
            a : integer;
        VAR  b : integer;

        b := a * a;

      PROCEND square;

MODEND example_cyb;
```

**Figure 9-2. Example of an EXAMPLE_CYB Source Listing**

## Preparing to Debug

After you create EXAMPLE_CYB, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAMPLE_CYB for use with Debug. You can then execute it under screen mode Debug control. Do this as follows:

1. EXAMPLE_CYB calls several file interface procedures that must be expanded through commands provided in the Source Code Utility (SCU) before the source code can be compiled. To do this, enter the following commands:

   ```
   /create_source_library
   /scu_create_deck deck=example_cyb modification=m1 ..
   ../source=$user.example_cyb
   /scu_expand_deck deck=example_cyb ..
   ../alternate_base=$system.cybil.osf$program_interface ..
   ../compile=$user.compile
   ```

   or abbreviated,

   ```
   /cresl
   /scu_cred d=example_cyb m=m1 s=$user.example_cyb
   /scu_expd d=example_cyb ..
   ../ab=$system.cybil.osf$program_interface ..
   ../c=$user.compile
   ```

2. Prepare for screen mode debugging and compile EXAMPLE_CYB now contained in permanent file $USER.COMPILE for use with Debug by entering the following commands:

   ```
   /change_interaction_style style=screen
   /cybil input=$user.compile binary=lgo ..
   ../optimization_level=debug debug_aids=all
   ```

   or abbreviated,

   ```
   /chais s=s
   /cybil i=$user.compile b=lgo ol=debug da=all
   ```

3. Execute under control of Debug by entering the following command:

   ```
   /execute_task file=lgo debug_mode=on
   ```

   or abbreviated,

   ```
   /exet f=lgo dm=on
   ```

The EXAMPLE_CYB source module is displayed in the source window. Debug functions are displayed at the the bottom of the screen.

## Displaying Screen Mode Commands

The functions below are used to display helpful information about the Debugging environment:

| | |
|---|---|
| HELP | Displays the help window. Press a function key and a short explanation of the function's use appears in the Help window. |
| ZMIN | Used to display the source listing in the source window. |

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The help window is displayed.

2. Press each function key corresponding to the functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. This exits HELP and the help window is removed.

4. Press the ZMIN function key. The following message is displayed in the upper right hand corner of the screen:

   ```
   Enter compiler input file for EXAMPLE_CYB
   ```

5. Enter the source file name:

   ```
   $user.compile
   ```

   The EXAMPLE_CYB source listing is displayed in the source window. Also, some new functions are displayed at the bottom of the screen.

6. Press the HELP key. The help window is displayed again.

7. Press each function key corresponding to the new functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each new function is displayed in the help window.

8. Press RETURN. Exit HELP.

### Setting Breaks

It is often helpful to suspend program execution when debugging a program. The Debug device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks.

| Function | Result |
|---|---|
| BKW | Scrolls backward to the previous screen of text. |
| FWD | Scrolls forward to the next screen of text. |
| LOCATE | Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text. |
| SETBRK | Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the first statement on the line containing the break. |

Perform the following steps to place three execution breaks in EXAMPLE_CYB:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_CYB:

   ```
   WHILE
   ```

   The cursor is moved to the line:

   ```
   WHILE month_row < 6 DO
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line:

   ```
   month_row := month_row + 1;
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_CYB source listing is displayed. Use the down-arrow key to position the cursor on that line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Press the FWD function key. The next screen of the EXAMPLE_CYB source listing is displayed in the source window.

7. Use the down-arrow key to move the cursor to the line:

   ```
   quotient := dividend / divisor;
   ```

8. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

9. Press the BKW key two times. The first screen of the EXAMPLE_CYB source listing is displayed in the source window.

## Debugging Test1

Using Debug, you can execute a program one statement or several statements at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

| | |
|---|---|
| CHAVAL | Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value. |
| GOTO | Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line. |
| HSPEED | Executes a program until a break is encountered or the program ends. |
| SEEVAL | Prompts you to enter a variable name, then displays the value of the variable in the output window. |
| STEP1 | Executes a program one statement at a time. |
| STEPN | Executes N statements of a program, where N is an integer. |

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEP1 function key. The statement:

   ```
   lfn := '$OUTPUT';
   ```

   is executed; the execution arrow now points to the statement:

   ```
   amp$open(lfn,amc$record,NIL,o,s);
   ```

2. Press the STEP1 function key again. The amp$open procedure is executed; moving the execution arrow to the first executable line in TEST1:

   ```
   FOR cntr := 1 TO 10 DO
   ```

3. Press the STEP1 function key seven times. An iteration of FOR loop is executed one statement at a time. The output from the iteration is displayed in the output window.

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

   ```
   cntr
   ```

   The value of CNTR is displayed in the output window:

   ```
   cntr = 2
   ```

   Thus, you can use SEEVAL to examine the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

   ```
   cntr=8
   ```

   The value of CNTR is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   cntr
   ```

   The following message is displayed in the output window:

   ```
   cntr = 8
   ```

   Thus, the change of value for CNTR is verified.

7. Press the STEPN function key. In the upper right hand corner of the screen, you are prompted for the number of lines to execute; enter:

   ```
   6
   ```

   STEPN executes 6 lines. The output from this loop iteration is displayed in the output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   cntr
   ```

   The value of COUNTER is displayed in the output window:

   ```
   cntr = 3
   ```

   Only the value of CNTR passed to the SQUARE call was changed. The value of a FOR loop control variable cannot be changed once the loop has been entered. Therefore, the value of CNTR used by this FOR loop remains unchanged.

9. Use the up-arrow key to move the cursor to the line:

   ```
   FOR cntr := 1 TO 10 DO
   ```

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes at this line.

11. Press the HSPEED function key. Execution resumes from the FOR statement. Since the FOR loop is executed anew, CNTR is initialized to 1. Execution of EXAMPLE_CYB continues until an execution break is encountered.

## Debugging Test2

After program execution is resumed in step 12 of TEST1, execution stops at the break set on the first statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

| | |
|---|---|
| BKW | Scrolls backward to the previous screen of text. |
| CHAVAL | Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value. |
| DELBRK | Deletes execution breaks. |
| FWD | Scrolls forward to the next screen of text. |
| HSPEED | Executes a program until a break is encountered or the program ends. |
| SEEVAL | Prompts you to enter a variable name, then displays the value of the variable in the output window. |

This section also uses the following features:

| | |
|---|---|
| HOME | Press the HOME key to move the cursor to the HOME line. line mode Debug commands can be entered on the HOME line for execution in screen mode Debug. |
| DISPLAY_ PROGRAM- _ VALUE | A line mode Debug command that displays the values of program variables. |

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the WHILE loop; output from the loop is displayed in the output window.

2. Press the HSPEED function key again. One iteration of the WHILE loop is executed; execution stops at the break set in the WHILE loop again. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   The following message is displayed in the ounput window:

   ```
   month_row = 2
   ```

5. Press the CHAVAL function key. When you are prompted, enter:

   ```
   month_row=4
   ```

6. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   The following message is displayed in the output window:

   ```
   month_row = 4
   ```

   Thus, the change to MONTH_ROW is verified.

7. Press the HSPEED function key. One iteration of the WHILE loop is executed.

8. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   the following message is then displayed in the output window:

   ```
   month_row = 5
   ```

   The value given to MONTH_ROW in step 5 is used by the WHILE loop.

9. Press the HOME key. The cursor moves to the HOME line.

10. Enter the line mode Debug command:

    ```
    display_program_value name=$all
    ```

    The values of all variables declared in EXAMPLE_CYB are displayed in the output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the HOME line. For more information about using line mode Debug commands see the Debug Usage Manual.

11. Press the DELBRK key. The execution break is deleted.

12. Press the down-arrow key until the cursor is inside of the output window.

13. Press the BKW key. The data in the output window scrolls backward. When the cursor is contained within the output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window.

14. Press the HSPEED function key. The execution of EXAMPLE_CYB resumes, stopping at the next break. The execution arrow points to the first statement in TEST3.

## Debugging Test3

After resuming execution of EXAMPLE_CYB in step 14 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an execution error is encountered:

| | |
|---|---|
| CHAVAL | Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value. |
| GOTO | Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line. |
| SEEVAL | Prompts you to enter a variable name, then displays the value of the variable in the output window. |
| STEP1 | Executes a program one statement at a time. |
| QUIT | Used to leave Debug. |

Perform the following steps to finish the example:

1. Press the STEP1 function key. Execution halts, and the following message flashes in the top right hand corner of the screen:

   ```
   divide_fault
   ```

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   divisor
   ```

   The following message is displayed in the output window:

   ```
   divisor = 0.
   ```

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

   ```
   divisor=1.0
   ```

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   divisor
   ```

   The following text is displayed in the output window:

   ```
   divisor=1.00000000000000E+0000
   ```

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points to the DIVISION statement, so program execution resumes with this statement.

6. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the STEP1 function key two more times. The result of the DIVISION statement is displayed in the output window.

8. Press the STEP1 function key. EXAMPLE_CYB ends and the following message is displayed in the output window:

   ```
   DEBUG:  The status at termination was: NORMAL.
   ```

9. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your CYBIL programs. For more information about screen mode Debug and line mode Debug commands, see the Debug Usage manual.

# Part II. Common CYBIL Input/Output

# How to Use Common CYBIL I/O 10

This chapter explains how to use CYBIL I/O and describes the features and limitations that are unique to the NOS/VE implementation of CYBIL I/O.

# How to Use Common CYBIL I/O 10

## Introduction

The Common CYBIL Input/Output procedures (referred to as CYBIL I/O) allow a CYBIL program to use the input/output capabilities of NOS/VE, principally for reading and writing files. CYBIL I/O is not designed specifically for NOS/VE, but is standardized for use on several operating systems (NOS/VE, NOS, NOS/BE, VSOS, EOS, and APOLLO Aegis I/O systems).If CYBIL I/O were implemented on these systems, CYBIL programs that use CYBIL I/O procedures could execute on any of these operating systems with little or no modification. Currently, however, CYBIL I/O is only available for NOS/VE.

**NOTE**

Display screen interfaces and the more sophisticated input/output capabilities of NOS/VE are beyond the scope of Common CYBIL I/O. For these, refer to the CYBIL File Management manual, the CYBIL Sequential and Byte-Addressable Files manual, and the CYBIL Keyed-File and Sort/Merge Interfaces manual.

---

CYBIL I/O procedures may be used for either disk or terminal input/output, and with either disk or tape files.[1]

The CYBIL I/O procedures and data types are stored in the NOS/VE program interface; they can be used in a CYBIL program but are not part of the CYBIL language as such. In brief, the components are the following:

- CYBIL procedures and functions, both standard and NOS/VE-dependent.

- CYBIL constants, variables, and data-types used within the procedures and functions.

1. When using CYBIL I/O with tape files, note the following: tape marks cannot be read or written, and tape files cannot be read in reverse.

- Exception conditions issued by CYBIL I/O.

These components employ the basic elements of the CYBIL language described in Part I of this manual: constants, variables, types, functions, and procedures. For a description of the general format of these CYBIL elements, refer to Part I of this manual. Part II, CYBIL Input/Output, describes the specific components of CYBIL I/O and explains how to use them in CYBIL programs.

Part II is made up of the following chapters:

- Chapter 10, How to Use Common CYBIL I/O, explains how to use CYBIL I/O procedures in a CYBIL program and describes the features and limitations that are unique to the NOS/VE implementation of CYBIL I/O.

- Chapter 11, Opening, Closing, and Structuring Files, which describes the procedures for performing these activities.

- Chapter 12, Reading and Writing Files, describes each of the procedures for reading and writing files with CYBIL I/O, and contains examples of CYBIL programs using CYBIL I/O.

- Chapter 13, NOS/VE-Specific Procedures and Functions for CYBIL I/O, describes the procedures and functions that can only be used with the NOS/VE implementation of CYBIL I/O.

- Appendix J lists the constants and data types used by CYBIL I/O, and Appendix K lists the CYBIL I/O error messages.

## Using CYBIL I/O Procedures

Each CYBIL I/O procedure resides as an externally referenced (XREF) declaration in a deck on the following source library file:

$SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE

The XREF procedure declarations for the keyed-file calls described in the CYBIL Keyed-File and Sort/Merge Interfaces manual are stored as decks in the source library file $SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE.

To use a CYBIL I/O procedure, you must include the following statements in your CYBIL source program:[2]

- A Source Code Utility (SCU) *COPYC directive for copying the XREF declaration from the source library $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE.
- Statements that declare, allocate, and initialize actual parameter variables as needed.
- The procedure call statement.
- An IF statement that checks the procedure completion status returned in the procedure's status variable.

Figure 10-1 lists a source program that illustrates the use of these CYBIL I/O procedures.

2. In the rest of this chapter, the term procedure is used in the broader sense of a subroutine or set of instructions which can be executed by a single statement. It therefore refers to both procedures and functions as described in Part I.

```
MODULE example1;

{ Directive to copy the XREF procedure declaration.}

*copyc cyp$get_next_record

{ This procedure reads the next record from a file }
{ that was opened as a record file and returns a status }
{ record to the caller.}

PROCEDURE get_next_record
  (record_file: cyt$file;
  pointer_to_target: ^SEQ ( * );
  VAR number_of_cells_read: integer;
  VAR status: ost$status);


{ Procedure call statement }

CYP$GET_NEXT_RECORD (record_file, pointer_to_target,
  number_of_cells_read, status);

{ Status record check. }

IF NOT status.NORMAL THEN
  RETURN;
IFEND;


PROCEND get_next_record;
MODEND example1;
```

**Figure 10-1. Example of a CYBIL I/O Call**

The following paragraphs describe in greater detail the SCU directives and CYBIL statements required for using CYBIL I/O procedures.

## Copying Procedure Declaration Decks

To use a CYBIL I/O procedure in a CYBIL module, you must include in the module an SCU *COPYC directive to copy the procedure's CYBIL XREF declaration from the source library file $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE.

The deck containing the procedure declaration has the same name as the procedure. For example, the CYP$GET_NEXT_RECORD procedure is declared in a deck named CYP$GET_NEXT_RECORD.

The *COPYC directives begin in column one and specify the name of the deck to be copied. In figure 10-1, they follow the MODULE statement.

Regardless of how many times a procedure is called, you need only one *COPYC directive per procedure. (For more information about the *COPYC directive, see the SCL Source Code Management manual.)

Procedure declaration decks list the parameters and their valid CYBIL types, which must be listed on a call to a CYBIL I/O procedure. When a CYBIL program is being compiled, the parameters on the call to the procedure are verified with the parameters and parameter types listed in the procedure's XREF declaration. If they do not match, the program compilation fails. After the module in figure 10-1 is expanded and compiled, the XREF procedure declaration is included in the source listing.

For an example of a procedure declaration deck, refer to Extracting CYBIL Procedure Decks later in this chapter.

In chapters 11, 12, and 13, the parameters and each parameter's required type are listed in the individual description for each CYBIL I/O procedure. In addition, the parameter types for all CYBIL I/O procedures are listed alphabetically in Appendix J of this manual.

### Expanding a Source Program

Before you can compile a source program containing one or more CYBIL I/O XREF procedures, you must first expand your source program. (Expanding a program generates the source code to be compiled.) You can use the SOURCE_CODE_UTILITY (SCU) subcommands to do this, or you can use the SCL command EXPAND_SOURCE_FILE.

The following SCL statements illustrate the way to use the SCL EXPAND_SOURCE_FILE command to expand a source program.

```
/expand_source_file, file=my_program ..
../alternate_base=($system.cybil.osf$program_interface, ..
../$system.common.psf$external_interface_source)
```

The command writes the expanded text on the default file, COMPILE. You then compile the expanded program text with the following:

```
/cybil input=compile list=listing list_options=(r, a)
```

The EXPAND_SOURCE_FILE example, above, shows the steps required to expand the CYBIL source program contained in figure 10-1. Spelled out, the steps consist of the following:

- Specify the name of the file to be expanded on the FILE parameter.

- Specify the $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE file on the ALTERNATE_BASE parameter. This file contains the XREF procedure decks for all CYBIL I/O procedures. If the CYBIL program uses one of the keyed-file procedures, the file $SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE must also be specified on the ALTERNATE_BASE parameter. The file is then expanded, and the XREF decks named on *COPYC directives in the CYBIL module are copied into the expanded source program. By default, the name of this expanded source program is COMPILE.

- Call the CYBIL compiler to compile the source program on file COMPILE, and write a source listing on file LISTING. The list options available on the CYBIL statement are described in Part I of this manual.

## Calling a CYBIL I/O Procedure

A call to a CYBIL I/O procedure has the same format as any CYBIL procedure call:

```
procedure_name (parameter_list);
```

For more information on CYBIL procedure calls, see Part I of this manual.

### Parameter List

The parameter list provides the procedure with input values and the locations at which it is to store output values. You can specify an input value as the value itself or as a variable containing the value.

**NOTE**

All parameters on a procedure call are required. You must specify a value or variable for each parameter in the parameter list.

CYBIL performs type checking on the variables and values specified in the parameter list. It compares the parameters on the procedure call with the parameter types listed in the XREF procedure declaration. Therefore, to make a successful call to a CYBIL I/O procedure, the parameters on the procedure call must conform to the parameter types specified in the procedure declaration deck.

Type checking and the valid parameter types for CYBIL programs are discussed in Part I of this manual.

#### Extracting CYBIL Procedure Decks

As mentioned earlier, the procedure declaration decks for all CYBIL I/O procedures are contained in the $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE library file. You can display a particular deck in this file for your own information by using the SCU subcommand EXTRACT_DECK, which extracts a deck and writes it to a file. The contents of each deck are identical to the parameters and types listed in each procedure call description in this manual.

The following example shows how to extract the CYP$GET_NEXT_RECORD procedure declaration deck and display it at your terminal. By default, the extracted deck is written to file SOURCE.

```
/source_code_utility
sc/extract_deck deck=cyp$get_next_record ..
sc../ab=$system.cybil.osf$program_interface
sc/edit_file source

 PROCEDURE [XREF] cyp$get_next_record
   (record_file: cyt$file;
    pointer_to_target: ^SEQ ( * );
   VAR number_of_cells_read: integer;
   VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc CYT$FILE
*copyc OST$STATUS
*copyc CYE$EXCEPTION_CONDITIONS
?? POP ??
sc/quit
```

## Parameter Types

As indicated by the procedure declaration deck displayed above, a call to the CYP$GET_NEXT_RECORD procedure must specify four parameters in its parameter list.

- The first parameter must specify a file identifier of type CYT$FILE.
- The second must specify a pointer of type ^SEQ.
- The third must specify a variable of type integer.
- The fourth must specify a variable of type OST$STATUS.

## VAR Parameters

The procedure call descriptions in this manual and the XREF procedure declaration decks both contain parameters that have a VAR listed with the parameter. The VAR indicates that the parameter is treated as an output parameter by the procedure; that is, a value is returned to the parameter by the procedure.

For example, the VAR listed with each procedure's status parameter indicates that the procedure returns a value to the status parameter.

In the procedure declaration deck, the VAR precedes the parameter name.

```
      VAR status: ost$status;
```

In the procedure call descriptions in this manual, the VAR is listed with the parameter's type.

```
     status:  VAR of ost$status;
```

For more information on declaring and assigning values to variables, see Part I of this manual.

## Checking the Completion Status

The last parameter on a CYBIL I/O procedure call must be a status variable (type OST$STATUS). Unlike the status parameter on SCL commands, the status parameter on these CYBIL calls is required, not optional. When the procedure completes, NOS/VE returns the completion status of the procedure in the specified status variable.

The program should check the completion status returned immediately after the procedure call. If the NORMAL field of the status variable is TRUE, the procedure completed normally. If the NORMAL field is FALSE, the procedure completed abnormally.

For example, the following program fragment uses a status variable named STATUS. Immediately after the CYP$GET_NEXT_RECORD call, an IF statement checks the value of the boolean field of the status record (STATUS.NORMAL). If its value is FALSE (NOT STATUS.NORMAL), the procedure terminates.

```
cyp$get_next_record (record_file, pointer_to_target,
    number_of_cells_read, status);
IF NOT status.NORMAL THEN
    RETURN;
IFEND;
```

### Status Condition Codes

When the procedure terminates abnormally, NOS/VE returns additional information about the condition that occurred. The following fields of the record return this information when the NORMAL field is FALSE:

condition

Exception condition code that uniquely identifies the condition (integer of type OST$STATUS_CONDITION_CODE). Each code can be referenced by its constant identifier as listed in the NOS/VE Diagnostic Messages manual.

text

String record (type OST$STRING) containing additional information about the condition. The record has the following two fields:

size

Actual string length in characters (0 through 256).

value

Text string (256 characters).

**NOTE**

The text field does not contain the error message. It contains items of information that are inserted in the error message template if the message is formatted using this status variable.

If the NORMAL field of the status record is FALSE, the program determines its subsequent processing. For example, it might check for a specific condition in the CONDITION field or determine the severity level of the condition with an OSP$GET_STATUS_SEVERITY procedure call. (The CYBIL System Interface manual contains the description of OSP$GET_STATUS_SEVERITY and other condition processing calls.)

## System Naming Convention

All identifiers defined by the NOS/VE program interface use the system naming convention. The system naming convention requires that all system-defined CYBIL identifiers have the following format:

```
idx$name
```

| Field | Description |
|---|---|
| id | Two characters identifying the product that uses the identifier. The following are the product identifiers referenced in this manual: |

| Product Identifier | Product Function |
|---|---|
| AM | Access method. |
| CY | CYBIL Input/Output. |
| FS | File system. |
| OS | Operating system. |

| Field | Description |
|---|---|
| x | Character indicating the CYBIL element type identified. |

| x | Description |
|---|---|
| c | Constant. |
| d | Deck. |
| e | Error condition. |
| p | Procedure. |
| t | Type. |

| Field | Description |
|---|---|
| $ | The $ character indicates that Control Data defined the identifier. |
| name | A string of characters describing the purpose of the element represented by the identifier. |

For example, the identifier CYP$GET_NEXT_RECORD follows the system's naming convention. Its product identifier is CY, for CYBIL Input/Output. The P following the product identifier indicates that it is a procedure name. The string GET_NEXT_RECORD describes the purpose of the procedure.

## Procedure Call Description Format

Chapters 10 and 11 of this manual describe the CYBIL I/O procedures. Each description uses the following format and subheadings:

| | |
|---|---|
| Purpose | Brief statement of the procedure function. |
| Format | Procedure call format showing the parameter positional order followed by individual parameter descriptions. |
| Parameters | Descriptions of the parameters in the preceding format, including the parameter's valid CYBIL type. |
| Conditions | List of condition identifiers returned by the procedure. The list is not complete; only the conditions that are likely to be of interest to the procedure user are listed. |
| Remarks | If present, additional information about procedure processing. |

### Parameter Description Format

Within a procedure description, each parameter description states the parameter's function, its values, and its valid CYBIL type. Appendix I of this manual contains an alphabetical listing of all parameter types for the CYBIL I/O procedures described in this manual.

If the parameter type is a set of system-defined identifiers, the parameter description lists all possible identifiers in the set and their meanings.

If the variable type is a record, the parameter description describes each field in the record. It states the field's name, its function, and its type.

## Features Unique to NOS/VE

The remainder of this chapter describes the features and limitations of CYBIL I/O on NOS/VE.

### Copying Procedure Declaration Decks in Bulk

There are two ways to declare CYBIL I/O procedures in a CYBIL program. One way, already described, is to include the name of the procedure on the *COPYC directive, such as *COPYC GET_NEXT_RECORD. With this method, there must be a *COPYC directive for each procedure used in the program. But another way, also using the *COPYC directive, is to declare the name of a deck which in turn declares all the CYBIL I/O procedures for a certain type of file, such as record files. Instead of a *COPYC directive for each of the record file procedures, one *COPYC directive would declare all of the record file procedures. The names of these general declaration decks and the procedures they declare are listed below.

| In order to declare these: | Use this deck name: |
|---|---|
| CYBIL I/O types | CYT$CYBIL_INPUT_OUTPUT |
| All procedures applicable to binary files | CYD$BINARY_FILE |
| All procedures applicable to record files | CYD$RECORD_FILE |
| All procedures applicable to text files | CYD$TEXT_FILE |
| All procedures applicable to display files | CYD$DISPLAY_FILE |

For example, if a CYBIL program includes the following in its source code,

```
*COPYC CYD$BINARY_FILE
```

all the procedure declaration decks for the binary-file procedures described in chapters 12 and 13 are automatically copied into the source program during program expansion.

**NOTE**

This method of declaring the binary-file procedures is efficient only if most or all of those procedures will actually be used by the program. If only a few are needed, it is better to declare them individually, because the CYD$BINARY_FILE declaration performs a tremendous amount of copying and would result in the needless use of system resources.

## File Names

File names specified on the CYBIL I/O procedures which open files (such as CYP$OPEN_FILE) must conform to the naming conventions for NOS/VE, and are interpreted as file references. Within NOS/VE, file references include the path, cycle, and position of the file.

## Position of File When Opened

You can specify the position at which a file is opened in one of several ways: (listed in order of precedence)

- With the open_position record on the FILE_SPECIFICATIONS parameter of CYP$OPEN_FILE (described in chapter 11).
- In the file reference (that is, on the file name) passed to CYP$OPEN_FILE.
- With the SCL command SET_FILE_ATTRIBUTES.

The order of precedence is as follows:

1. If the open_position record on the FILE_SPECIFICATIONS parameter specifies a file position, that position is used when the file is opened.
2. If the open_position record does not specify a file position, then the file position included in the file reference on CYP$OPEN_FILE is used.
3. If the file reference does not include a file position, then the position specified on the SET_FILE_ATTRIBUTES command (if specified for this instance of attachment) is used.
4. If a file position has not been specified by the SET_FILE_ATTRIBUTES command for this instance of attachment, then the file's open position is beginning-of-information.

**NOTE**

If you want the file opened at beginning-of-information, and if the file was not explicitly attached (with the ATTACH_FILE command), then it is not necessary to specify any file position at all: beginning-of-information is automatically used when the file is opened with CYP$OPEN_FILE. If the file was explicitly attached, but no file position has been specified for this instance of attachment, it is likewise not necessary to specify any file position: beginning-of-information is automatically used.

---

For a description of the CYP$OPEN_FILE procedure and the open_position record of the FILE_SPECIFICATIONS parameter, refer to chapter 11.

## Position of File When Closed

With the FILE_POSITION parameter of CYP$CLOSE_FILE (described in chapter 11), the caller can specify where a file is positioned before it is closed. This position is retained after the file is closed only if all of the following are true:

- The file has been explicitly attached (with the ATTACH_FILE command).
- The close_file_disposition record specified on the FILE_SPECIFICATIONS parameter of CYP$OPEN_FILE is CYC$RETAIN_FILE.
- Subsequent instances of open within the job specify an open_position of CYC$ASIS.

The open_position and close_file_disposition records are explained in chapter 11, under File Specification Records.

## File Attributes

Because CYBIL I/O provides standard input/output interfaces for several operating systems, no provision is made to directly set or interrogate NOS/VE file attributes, except as described below.[3]

CYBIL I/O follows a simple set of rules for file attributes.

3. At present, CYBIL I/O has only been implemented for NOS/VE.

- If the file has never been opened, the file is a new file and CYBIL I/O defines file attributes as listed in tables 10-1 and 10-2. If the FILE_SPECIFICATIONS parameter of the CYP$OPEN_FILE call contains a value from which the attribute may be set, that value is used.

- If the file has been previously opened, CYBIL I/O considers the file an old file and does not modify or define any file attributes.

**Table 10-1. File Attributes for New Files: Binary and Record**

| File_Attribute | Binary Files | Record Files |
|---|---|---|
| file contents | CYC$UNKNOWN_ CONTENTS | CYC$UNKNOWN_ CONTENTS |
| file structure | CYC$UNKNOWN_ STRUCTURE | CYC$UNKNOWN_ STRUCTURE |
| file processor | CYC$UNKNOWN_ PROCESSOR | CYC$UNKNOWN_ PROCESSOR |
| page format[1] | CYC$BURSTABLE_ FORM | CYC$BURSTABLE_ FORM |
| page length[1] | 60 lines | 60 lines |
| page width[1] | 132 columns | 132 columns |

1. The attribute values for page format, page length, and page width are for NOS/VE files.

**Table 10-2. File Attributes for New Files: Text and Display**

| File_Attribute | Text Files | Display Files |
|---|---|---|
| file contents | CYC$LEGIBLE | CYC$LIST |
| file structure | CYC$UNKNOWN_ STRUCTURE | CYC$UNKNOWN_ STRUCTURE |
| file processor | CYC$UNKNOWN_ PROCESSOR | CYC$UNKNOWN_ PROCESSOR |
| page format[1] | CYC$CONTINUOUS_ FORM | CYC$BURSTABLE_ FORM |
| page length[1] | 60 lines | 60 lines |
| page width[1] | 132 columns | 132 columns |

1. The attribute values for page format, page length, and page width are for NOS/VE files.

You can define the page_length, page_width, page_format, file_contents, and file_processor attributes for new files on the FILE_SPECIFICATIONS parameter of the CYP$OPEN_FILE procedure, which is described in chapter 11.

In addition, file attributes may be defined via SCL commands or CYBIL procedures prior to calling CYP$OPEN_FILE. In this case, CYBIL I/O considers the file an old file and does not define or modify any of the permanent attributes.

## File Structure

Four kinds of files can be used with CYBIL I/O: binary, record, text, and display. All of these files have a beginning-of-information and an end-of-information. On NOS/VE, files can be further subdivided into partitions and records. (Binary files can only be subdivided into partitions.)

| Level | Description |
|---|---|
| Partition | A partition begins either at the beginning-of-information or after the end-of-partition of the previous partition. |
| Record | A record begins at the beginning-of-information, after an end-of-partition, or after the end-of-record of a preceding record. |

**NOTE**

Partitions should only be used when necessary: in certain reading, writing, and positioning operations, an end-of-partition can be mistaken for an end-of-information.

Although the end-of-information can only be implicitly created (the end-of-information follows the last item written on a file), it can be explicitly detected with the CYP$CURRENT_FILE_POSITION procedure, described in chapter 11.

For more information on the structure of each of the four types of files, refer to chapters 11 and 12.

# Opening, Closing, and Structuring Files 11

This chapter describes the procedures for opening, closing, and structuring files.

# Opening, Closing, and Structuring Files 11

# Introduction

This chapter describes the procedures for opening and closing files, positioning files, and for creating file structure. The procedures in this chapter may be used with any of the four types of files supported by CYBIL I/O:

| | |
|---|---|
| Binary | A binary file is treated as a stream of cells. Any further structure in the file is provided by the program that creates it. The file can be accessed either sequentially or randomly. Random access is made possible by file keys that mark cell addresses. (FILE_KEY is a parameter on the binary file procedures described in chapter 12.) The file can be positioned to beginning-of-information or end-of-information, or to any file key. |
| Record | A record file is a sequence of logical records. A record can be read or written as a complete unit or in pieces (as "partial" reads or writes). A record file can only be accessed sequentially. It can be positioned to beginning-of-information, end-of-information, or forward or backward a specified number of records or partitions. |
| Text | A text file is essentially a file of records. Each record is treated as a line of characters, and each end-of-record as an end-of-line. A text file can only be accessed sequentially. It may be positioned to beginning-of-information or end-of-information, and an output text file can be tabbed to a specified column or skipped a specified number of lines. |
| Display | A display file is a write-only text file, for printing, displaying at a terminal, or sending to any device that uses format control characters. Format control operations are possible, such as limiting the number of printed lines on a page, positioning the next line at a specified line number, or overprinting a line. There are also several ways of handling page-overflow. |

These four types of files are described in greater detail in chapter 12, along with the CYBIL I/O procedures for reading and writing files.

The CYBIL I/O procedures and functions are not defined in the

CYBIL language itself; they are part of the NOS/VE program interface. For instance, to use a CYBIL I/O procedure, a CYBIL program must include a *COPYC directive for the procedure declaration deck containing that procedure. For more information on referencing procedure declaration decks and on expanding CYBIL source programs, refer to chapter 10, How to Use CYBIL I/O.

The procedures and functions described in this chapter are the following:

CYP$OPEN_FILE

Opens a file.

CYP$CLOSE_FILE

Closes a file.

CYP$POSITION_FILE_AT_BEGINNING

Positions a file to its beginning-of-information.

CYP$POSITION_FILE_AT_END

Positions a file to its end-of-information.

CYP$CURRENT_FILE_POSITION

Returns the current position of a file.

CYP$LENGTH_OF_FILE

Returns the length of a file.

CYP$WRITE_END_OF_RECORD

Writes an end-of-record on a record file.

CYP$WRITE_END_OF_PARTITION

Writes an end-of-partition on a file.

CYP$OPERATING_SYSTEM

Returns a value that identifies the operating system on which a program is running.

## CYP$OPEN_FILE

**Purpose** Opens a file.

**Format** **CYP$OPEN_FILE (file_name, file_specifications, file, status)**

**Parameters** **file_name**: cyt$file_name;

The name of the file to be opened. On NOS/VE, a file name may be up to 512 characters in length, and may be a file reference.

**file_specifications**: cyt$file_specifications;

Pointer to an array of case-variant records. The array must be initialized before the parameter can be passed. The values specified in the records determine how the file is to be used.

Any record left unspecified will default to the value in the list below. If a NIL value is specified, all of the records take the default values.

| **file_specification record** | **Default** |
|---|---|
| close_file_disposition | CYC$DEFAULT_FILE_DISPOSITION |
| file_access | CYC$READ_WRITE |
| file_character_set | CYC$ACSII |
| file_existence | CYC$NEW_OR_OLD_FILE |
| file_kind | CYC$RECORD_FILE |
| new_page_procedure | CYC$OMIT_PAGE_PROCEDURE |
| page_format | CYC$BURSTABLE_FORM |
| open_position | CYC$DEFAULT_OPEN_POSITION |
| page_length | system dependent |
| page_width | system dependent |

| | |
|---|---|
| file_contents | CYC$UNKNOWN_ CONTENTS |
| file_processor | CYC$UNKNOWN_ PROCESSOR |

Following the description of CYP$OPEN_FILE is more information about the file specification records, the values that may be specified, and their defaults. For examples demonstrating the use of this parameter and the CYP$OPEN_FILE procedure in CYBIL programs, refer to the program examples in chapter 12.

**file**: VAR of cyt$file;

Returns a pointer that must be used on all other calls to the file specified by the FILE_NAME parameter. This pointer is an identifier defined when the file is opened with this procedure. Until the file is closed (with the CYP$CLOSE_FILE procedure), all references to the file must include this identifier. In other words, this identifier remains defined until it is passed to the CYP$CLOSE_ FILE procedure. (This particular instance of the file being opened, with this identifier, is called the "instance of open" for the file.)

Attempting to call a CYBIL I/O procedure with an altered or undefined pointer will have unpredictable results.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**

cye$file_already_exists
cye$file_not_found
cye$incorrect_open_request
cye$no_memory_to_open_file

**Remarks**

- The length of the file name and the characters included in the file name must conform to the requirements of the operating system; otherwise, the open will be aborted and abnormal status will be returned.

- The values entered for the FILE_SPECIFICATIONS records determine how the file is to be opened, how it is to be operated upon, and what is to be done with the file after it is closed. For more information on how the FILE_SPECIFICATIONS records are built, refer to the discussion of file specifications in this chapter, and to Appendix I.

- If the capabilities of the FSP$OPEN_FILE procedure are needed in opening the file, refer to chapter 13 for a CYBIL I/O file-opening procedure that passes values directly to FSP$OPEN_FILE. (The FSP$OPEN_FILE procedure is described in the CYBIL File Management manual.)

## File Specification Records

The file specification records are used by the CYP$OPEN_FILE procedure to determine how the file is to be opened, how it is to be operated upon, and what is to be done with it when it is closed. Each is specified as a value in a record in an initialized array; the array is then named on the FILE_SPECIFICATIONS parameter in the call to CYP$OPEN_FILE.

The example programs in chapter 12 show how file specifications are established in CYBIL programs. Additional information about file specifications is in the description of the CYP$OPEN_FILE procedure earlier in this chapter, and in Appendix J, which lists all the CYBIL I/O constants and types.

The following are the file specification records established when a file is opened, and the values allowed for each.

*close_file_disposition*

Determines whether or not a file is detached (or deleted) after it is closed (type CYT$CLOSE_FILE_DISPOSITION).

The following values are available:

| Disposition | Result |
|---|---|
| CYC$UNLOAD_FILE or CYC$RETURN_FILE or CYC$DETACH_FILE | An explicit detach is performed when the file is closed, provided it has no other instances of open outstanding in the job. |
| CYC$RETAIN_FILE | If the file was explicitly attached prior to open, the file remains attached. |
| CYC$DELETE_FILE | If the file is local, it is detached; if permanent, it is deleted. |
| CYC$DEFAULT_FILE_DISPOSITION | If the file was implicitly attached by CYP$OPEN_FILE and the file has no other instances of open outstanding in the job, the file is detached when it is closed. |

The default value is CYC$DEFAULT_CLOSE_DISPOSITION.

*file_access*

Specifies the modes of access permitted on the file's data (type CYT$FILE_ACCESS).

When the file is opened, the file_access record serves to validate all read/write requests to the file for the instance of open. The attempt to write to a file opened for read or to read from a file opened for write will be blocked and abnormal status will be returned. Enter one of the following values:

| | |
|---|---|
| CYC$READ | Read-only access. |
| CYC$WRITE | Write-only access. |
| CYC$READ_WRITE | Read or write access. |

The default value is CYC$READ_WRITE.

**NOTE**

New files must be opened with CYC$WRITE or CYC$READ_WRITE. If a file is opened as a new file (CYC$NEW_FILE on the file_existence record) with CYC$READ, the attempt to open the file will fail and abnormal status will be returned.

*file_existence*

Specifies whether the file is created when it is opened (type CYT$FILE_EXISTENCE). Enter one of the following values:

| Constant | Description |
|---|---|
| CYC$OLD_FILE | The file already exists, otherwise the file open procedure returns abnormal status. |
| CYC$NEW_FILE | The file is new and is created by this instance of open. CYP$OPEN_FILE returns abnormal status if this value is specified and the file already exists. |
| CYC$NEW_OR_OLD_FILE | If the file does not exist it will be created. |

The default value is CYC$NEW_OR_OLD_FILE.

*file_kind*

Specifies the file types, and thus limits the kinds of CYBIL I/O procedures that may be addressed to a file (type CYT$FILE_KIND). The file types are the following:

CYC$BINARY

CYC$RECORD

CYC$TEXT

CYC$DISPLAY

The default value is CYC$RECORD_FILE. These four kinds of files are described at the beginning of this chapter and in chapter 12.

For example, if a file is opened as a text file, the attempt to use any record, binary, or display file procedure calls is prohibited and the status variable returned indicates CYE$INCORRECT_OPERATION.

*file_character_set*

Specifies the character set for text and display files (type CYT$FILE_CHARACTER_SET).

The only character set supported by NOS/VE is CYC$ASCII (8-bit ASCII code), which is the default value.

**NOTE**

The file_character_set is used only by text and display-type files. If this record is defined for binary or record files, it is ignored.

*file_contents*

Describes the contents of a file (type CYT$FILE_CONTENTS). The use of this value is system dependent.

The following are the available values:

CYC$ASCII_LOG

CYC$BINARY

CYC$BINARY_LOG

CYC$DATA

CYC$FILE_BACKUP

CYC$LEGIBLE

CYC$LEGIBLE_DATA

CYC$LEGIBLE_LIBRARY

CYC$LEGIBLE_UNKNOWN

CYC$LIST

CYC$LIST_UNKNOWN

CYC$OBJECT

CYC$OBJECT_DATA

CYC$OBJECT_LIBRARY

CYC$SCREEN

CYC$SCREEN_FORM

CYC$UNKNOWN_CONTENTS

The default value is CYC$UNKNOWN_CONTENTS.

*file_processor*

Describes the file processor (type CYT$FILE_PROCESSOR). The use of this value is system-dependent.

The following are the available values:

CYC$ADA

CYC$APL

CYC$ASSEMBLER

CYC$BASIC

CYC$C

CYC$COBOL

CYC$CYBIL

CYC$DEBUGGER

CYC$FORTRAN

CYC$LISP

CYC$PASCAL

CYC$PLI

CYC$PPU_ASSEMBLER

CYC$PROLOG

CYC$SCL

CYC$SCU

CYC$UNKNOWN_PROCESSOR

CYC$VX

The default value is CYC$UNKNOWN_PROCESSOR.

*new_page_procedure*

Specifies how page-overflow conditions are handled for display-type files (type CYT$NEW_PAGE_PROCEDURE). This specification builds a record of type CYT$PAGE_PROCEDURE_KIND. For the tag field of the record, specify CYC$USER_SPECIFIED_PROCEDURE, CYC$STANDARD_PROCEDURE, or CYC$OMIT_PAGE_PROCEDURE, described below.

CYC$USER_SPECIFIED_PROCEDURE

Whenever a page overflow condition occurs, CYBIL I/O automatically calls the procedure specified by the user_procedure field. The user_procedure field (type CYT$USER_PAGE_PROCEDURE) is a pointer to the user's page-overflow procedure. It passes three parameters to that procedure:

**display_file**: cyt$file

File ID established when the display file was opened.

**next_page_number**: integer

Page number of the overflow page.

**status**: ost$status

Status variable in which the status value is returned.

CYC$STANDARD_PROCEDURE

Whenever a page overflow condition occurs, CYBIL I/O automatically initiates a display-page eject and produces a standard title-line followed by one blank line. The title field of the new_page_procedure record specifies a string of characters that CYBIL I/O will include in the standard title line. (For a description of the standard title line, refer to Page-Overflow Processing for Display Files in chapter 12.)

CYC$OMIT_PAGE_PROCEDURE

Causes a display-page eject.

The default value is CYC$OMIT_PAGE_PROCEDURE.

**NOTE**

New_page_procedure is used only by display files. If this record is defined for any other type of file, it is ignored.

*page_format*

Specifies the presence and frequency of titling in a display file with file contents of CYC$LIST or CYC$LIST_UNKNOWN (type CYT$PAGE_FORMAT). (Titling is determined by the new_page_procedure record, explained above.)

The following values are available:

| Constant | Resulting Page Format |
|---|---|
| CYC$BURSTABLE_FORM | Titling and display-page eject occur at the frequency defined by the page length of the file. This is the recommended value for files that are to be listed on a forms printer with a page eject required for each page. |
| CYC$NON_BURSTABLE_FORM | Titling is separated from other data by a triple space rather than by forcing a display-page eject as in CYC$BURSTABLE_FORM. A display-page eject and titling also occur at the frequency defined by the page length of the file. |
| CYC$CONTINUOUS_FORM | Titling appears once at the beginning of the file followed by triple spacing. |
| CYC$UNTITLED_FORM | No titling and no display-page-eject occur anywhere in the file. |

**NOTE**

Page_format is used only by display files. If this record is defined for any other type of file, it is ignored.

*open_position*

Designates where the file should be initially positioned when it is opened (type CYT$OPEN_CLOSE_POSITION). For an explanation of how this record relates to other means of specifying file position when the file is opened, refer to Position of File When Opened, in chapter 10.

The following values are available:

CYC$BEGINNING

The file is opened at beginning-of-information. (CYC$BEGINNING takes precedence over any other file position specification, such as file position specified in the file reference.)

CYC$END

The file is opened at end-of-information. (CYC$END takes precedence over any other file position specification, such as file position specified in the file reference.)

CYC$ASIS

If the file has been explicitly attached, and if CYC$RETAIN_FILE was the close_file_disposition when the file was previously closed within this instance of attachment, the file is positioned to whatever was specified as the file position when it was closed. If the file was not explicitly attached, or if its close_file_disposition record when closed was not CYC$RETAIN_FILE, the file is opened at its beginning-of-information.

CYC$DEFAULT_OPEN_POSITION

If the open position was specified in the file reference, or was specified on the SCL command SET_FILE_ATTRIBUTES, CYP$OPEN_FILE uses that position when opening the file. If a file is opened with no file position specified at all, its open position is beginning-of-information.

The default value is CYC$DEFAULT_OPEN_POSITION.

If you want the file opened at beginning-of-information, and if the file was not explicitly attached (with the ATTACH_FILE command), then it is not necessary to specify any file position at all: beginning-of-information is automatically used when the file is opened with CYP$OPEN_FILE. If the file is explicitly attached, but no file position has been specified for this instance of attachment, it is likewise not necessary to specify any file position: beginning-of-information is automatically used.

**NOTE**

The FILE_POSITION parameter on the CYP$CLOSE_FILE procedure is also of type CYT$OPEN_CLOSE_POSITION. Under certain circumstances, this parameter determines the file's position at its close. For more information on this parameter, refer to the description of the CYP$CLOSE_FILE procedure later in this chapter.

*page_length*

Specifies the number of lines on a page for display-type files (type CYT$PAGE_LENGTH). The page length may be 1 to 439,804,651,103 lines. The constant CYC$PAGE_LIMIT specifies the maximum.

The default is a system-dependent value (60 lines on NOS/VE).

**NOTE**

Page_length is used only by display files. If this record is defined for any other type of file, it is ignored.

*page_width*

Specifies the maximum length of a text line for display or text files (type CYT$PAGE_WIDTH). The length may be 1 to 65,535 characters. The following values are available:

| Constant | Line Length in Characters |
|---|---|
| CYC$NARROW_PAGE_WIDTH | 80 |
| CYC$WIDE_PAGE_WIDTH | 132 |
| CYC$MAX_PAGE_WIDTH | 65535 |

The default is a system-dependent value (CYC$WIDE_PAGE_WIDTH on NOS/VE).

**NOTE**

Page_width is used only by display and text files. If this record is defined for any other type of file, it is ignored.

## CYP$CLOSE_FILE

| | |
|---|---|
| **Purpose** | Closes a file. |
| **Format** | **CYP$CLOSE_FILE (file, file_position, status)** |

**Parameters**

**file**: cyt$file;

File identifier established when the file was opened.

**file_position**: cyt$open_close_position;

This parameter specifies where the file is positioned before it is closed, but only if the file was explicitly attached (with the SCL command ATTACH_FILE) before it was opened with CYP$OPEN_FILE. If the file was not explicitly attached, this parameter is ignored.

Enter one of the following values:

CYC$BEGINNING

File is rewound to beginning-of-information and then closed.

CYC$END

File is positioned to end-of-information and then closed.

CYC$ASIS

File is closed without positioning.

CYC$DEFAULT_OPEN_POSITION

Same as CYC$ASIS.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**

cye$file_not_open
cye$incorrect_input_request

**Remarks**

- The close_file_disposition record of the file specifications that were established when the file was opened will determine what happens to the file when it is closed (whether it is retained, returned, unloaded, or deleted). The close_file_disposition record is explained earlier in this chapter under File Specification Records.

- The value of the FILE_POSITION parameter determines the file's position at its close only if the file was explicitly attached before it was opened. Furthermore, the value of this parameter is used only if the value of the close_file_disposition record of the file specifications was CYC$RETAIN_FILE, and if subsequent instances of open within the job specify CYC$ASIS on the FILE_POSITION parameter at each close.

- No matter where file position is specified (on the open_position record of FILE_SPECIFICATIONS, in the file referecne, on the SET_FILE_ATTRIBUTES command, or on the FILE_POSITION parameter of CYP$CLOSE_FILE), that specification does not become a permanent attribute of the file. It is in effect only during the instance of open or instance of attachment.

## Positioning Files

This section describes the procedures for positioning files: CYP$POSITION_FILE_AT_BEGINNING and CYP$POSITION_FILE_AT_END. It also describes the functions for checking the current position of the file, CYP$CURRENT_FILE_POSITION, and for checking the file's length, CYP$LENGTH_OF_FILE.

In addition, there are several ways of specifying where the file is to be positioned when it is opened. These are described in chapter 10, under Position of File at Open, and in this chapter under the CYP$OPEN_FILE procedure.

## CYP$POSITION_FILE_AT_BEGINNING

**Purpose** Positions a file at its beginning-of-information.

**Format** **CYP$POSITION_FILE_AT_BEGINNING (file, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open

**Remarks** To use this procedure, the file must have been opened with the CYP$OPEN_FILE procedure.

## CYP$POSITION_FILE_AT_END

**Purpose** Positions a file at its end-of-information.

**Format** **CYP$POSITION_FILE_AT_END (file, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request

**Remarks** To use this procedure, the file must have been opened with the CYP$OPEN_FILE procedure.

## CYP$CURRENT_FILE_POSITION Function

**Purpose** Returns the current position of a file.

**Format** **CYP$CURRENT_FILE_POSITION** (**file**): cyt$current_file_position;

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**Remarks**

- The following are the values that may be returned by this function:

  CYC$BEGINNING_OF_INFORMATION

  File is at beginning-of-information.

  CYC$MIDDLE_OF_RECORD

  File is at middle-of-record.

  CYC$END_OF_RECORD

  File is at end-of-record.

  CYC$END_OF_PARTITION

  File is at end-of-partition.

  CYC$END_OF_INFORMATION

  File is at end-of-information.

- Following any type of read or positioning operation, this function returns the current file position. Following most types of write operations, this function will return CYC$END_OF_INFORMATION. If the previous operation was a write to a binary file, this function returns CYC$MIDDLE_OF_RECORD unless the write extended the length of the file, in which case the function returns CYC$END_OF_INFORMATION.

- This function returns CYC$MIDDLE_OF_RECORD following a read from a binary file unless the NUMBER_OF_CELLS_READ parameter on the CYP$GET_NEXT_BINARY or CYP$GET_KEYED_BINARY procedure returned a value of 0 (zero). In this case, CYP$CURRENT_FILE_POSITION would return CYC$END_OF_PARTITION or CYC$END_OF_INFORMATION to indicate which file boundary condition was encountered.

## CYP$LENGTH_OF_FILE Function

**Purpose** Returns the length of a file. (The length is the number of cells in the file.)

**Format** **CYP$LENGTH_OF_FILE (file)**: integer;

**Format** **file**: cyt$file;

File identifier established when the file was opened.

## Creating File Structure

On NOS/VE, CYBIL I/O supports two levels of file-subdivision: records and partitions.

This section describes the CYP$WRITE_END_OF_RECORD procedure for creating end-of-records, and the CYP$WRITE_END_OF_PARTITION procedure for creating an end-of-partition.

In addition to CYP$WRITE_END_OF_RECORD, there are other procedures which create end-of-records. These procedures (which are described in chapter 12) are the following:

- CYP$WRITE_END_OF_LINE.
- CYP$PUT_NEXT_RECORD and CYP$PUT_NEXT_LINE.
- CYP$PUT_PARTIAL_RECORD with its LAST_PART_OF_RECORD parameter set to TRUE.
- CYP$PUT_PARTIAL_LINE with its LAST_PART_OF_LINE parameter set to TRUE.

Although the end-of-information can only be implicitly created (it follows the last item physically written on a file), it can be explicitly detected with the CYP$CURRENT_FILE_POSITION function.

## CYP$WRITE_END_OF_RECORD

**Purpose** Writes an end-of-record to a record file.

**Format** **CYP$WRITE_END_OF_RECORD (record_file, status)**

**Parameters** **record_file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the last write to the file was partial, that record is completed; otherwise, an empty record results.
- Attempting to use this procedure on a file not opened as a record-type file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened with either write access or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$WRITE_END_OF_PARTITION

**Purpose** Writes an end-of-partition in a file.

**Format** **CYP$WRITE_END_OF_PARTITION (file, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request

**Remarks**

- If the last write to the specified file was a partial write (by means of the CYP$PUT_PARTIAL_RECORD or CYP$PUT_PARTIAL_LINE procedure), the record or line is terminated before the end-of-partition is written.
- Attempting to use this procedure with a file not opened with either write access or read write access will return CYC$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$OPERATING_SYSTEM Function

**Purpose** Returns a value that identifies the operating system on which a program is running.

**Format** **CYP$OPERATING_SYSTEM**: cyt$system_type;

**Remarks**

- This function allows a program to handle any operating-system dependencies by first checking the identity of the operating system. (At present, the value of this function is somewhat limited, since CYBIL I/O has only been implemented for NOS/VE.)
- On NOS/VE, this function returns the value CYC$NOSVE.

# Reading and Writing Files 12

This chapter describes the CYBIL I/O procedures and functions for reading and writing files.

# Reading and Writing Files 12

This chapter describes the CYBIL I/O procedures and functions for reading and writing files. These procedures are designed to work in CYBIL programs run on other operating systems, such as NOS, without modifications.[1]

The procedures specifically designed only for use on CYBIL for NOS/VE are described in chapter 13. These procedures have features and limitations unique to NOS/VE, and are not intended to work in CYBIL programs run on other systems.

As explained in chapter 10, CYBIL I/O procedures and functions are not defined in the CYBIL language itself; they are part of the NOS/VE program interface. For more information on referencing procedure declaration decks and on expanding CYBIL source programs, refer to chapter 10, How to Use Common CYBIL I/O.

Each of the procedures in this chapter can only be used with one of the four file-types: binary, record, text, or display. (Some of the procedures can be used with both text and display files.) For each file-type there is a section in this chapter describing the procedures and functions which pertain only to that type. In addition, each file-type has certain characteristics and limitations of its own. These are also discussed in the section on that file-type.

1. At present, CYBIL I/O is only available on NOS/VE.

## Binary Files

The procedures and functions described in this section are for use with binary files only.

To read or write a file using CYBIL I/O, the CYBIL type of the parameter specifying the data to be read or written must match the CYBIL type of the program variable containing the data to be read or written. Moreover, the CYBIL I/O binary file procedures require that the data be specified as a pointer to a CYBIL sequence. Programs using the binary file procedures must therefore specify the data as a variable of type pointer to CYBIL sequence. This pointer is usually defined by using the CYBIL #SEQ function.

For example, given the following CYBIL variable declarations:

```
VAR
  data_item_1: my_data_type,
  data_item_2: ^my_data_type,
  data_item_3: ^array [1 .. 50] of my_data_type;
```

pointers to CYBIL sequences may be defined as follows:

```
#SEQ (data_item_1)
#SEQ (data_item_2^)
#SEQ (data_item_3^)
#SEQ (data_item_3^ [5])
```

There are examples of the #SEQ function under Program Examples Using Binary Files later in this chapter.

### Binary File Structure

CYBIL I/O imposes no structure on the data in a binary file. Therefore, any structure to be found in a binary file must be provided for and interpreted by the user program. For instance, the task that writes data on a binary-type file is responsible for determining how the data is to be read. It should write data-organization indicators as needed. As a result, a program that reads the binary file data must use the data conventions imposed by the program that wrote the data.

CYBIL I/O treats the data in a binary file as a sequence of cells. Calls to the binary read and write procedures result in a mapping of cells between the file and the CYBIL program variable.

Binary files may be subdivided into partitions but not into records.

Binary files may be read and written in either a random or sequential manner. Random access of binary files is possible via the FILE_KEY parameter on the binary file procedures described in this section. The file key may be viewed as an offset pointer that marks cell addresses within a binary file. The file keys identify the number of the cell within the file at which the data transfer begins.

**NOTE**

CYBIL I/O does not maintain a directory of file keys for binary files. It is the user's responsibility to create and maintain any directories that may be required.

---

When a binary file is opened, the file key is undefined. To access the file using the CYP$PUT_KEYED_BINARY and CYP$GET_KEYED_BINARY procedures, the file key must first be set to the current (open) position of the file. You can use the CYP$GET_NEXT_BINARY procedure or the CYP$BINARY_FILE_KEY function to get the address of the current file position.

The sequential access procedures transfer data to or from the "address" or file key at which the file is currently positioned. As with record files, the data read or written is transferred as a block of cells that are mapped to the CYBIL data structure being read or written.

Binary files may be positioned to the beginning-of-information, end-of-information, or to any file key within the file. Because binary files can be accessed randomly, positioning a binary file at the beginning-of-information and writing to the file does not necessarily mean that existing data (which follows the data being written) will be lost. (The opposite is true of record files, which are described later in this chapter).

The following procedures and functions may be used with binary files only. These procedures and functions are described in greater detail on the following pages.

CYP$GET_NEXT_BINARY

Reads data from a binary file.

CYP$GET_KEYED_BINARY

Reads data from a binary file.

CYP$PUT_NEXT_BINARY

Writes data to a binary file.

CYP$PUT_KEYED_BINARY

Writes data to a binary file.

CYP$POSITION_BINARY_AT_KEY

Positions a binary file to a specified file cell address.

CYP$BINARY_FILE_KEY

Returns the file cell address at which a binary file is currently positioned.

## CYP$GET_NEXT_BINARY

**Purpose** Reads data from a binary file.

**Format** **CYP$GET_NEXT_BINARY (binary_file, pointer_to_target, file_key, number_of_cells_read, status)**

**Parameters** **binary_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_target**: ^SEQ ( * );

The data structure into which data is to be read.

**file_key**: VAR of integer;

Returns the file cell address from which the read began.

**number_of_cells_read**: VAR of integer;

Returns the number of cells actually read. The value returned is normally the size of the data structure referenced by the POINTER_TO_TARGET parameter.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation

**Remarks**

- The data is read from the current position of the file.
- If end-of-partition or end-of-information is detected during a read, the NUMBER_OF_CELLS_READ parameter returns only the cells read before the end-of-partition or end-of-information was detected.
- Attempting to use this procedure on a file not opened as a binary-type file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

## CYP$GET_KEYED_BINARY

**Purpose** Reads data from a binary file.

**Format** **CYP$GET_KEYED_BINARY (binary_file, pointer_to_target, file_key, number_of_cells_read, status)**

**Parameters** **binary_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_target**: ^SEQ ( * );

The data structure into which data is to be read.

**file_key**: integer;

The file cell address at which the read is to begin.

**number_of_cells_read**: VAR of integer;

Returns the number of cells actually read. The value returned is normally the size of the data structure referenced by the POINTER_TO_TARGET parameter.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**

cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation
cye$key_past_eoi

**Remarks**

- If ~~end-of-partition or end-of-information is detected~~ during a read, the NUMBER_OF_CELLS_READ parameter returns only the cells read before the end-of-partition or end-of-information was detected.

- If the FILE_KEY parameter specifies a cell beyond the end-of-information, no data is read, CYP$GET_KEYED_BINARY will return CYE$KEY_PAST_EOI in the status variable, and the position of the file remains unchanged.

- Attempting to use this procedure on a file not opened as a binary-type file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

## CYP$PUT_NEXT_BINARY

**Purpose** Writes data to a binary file.

**Format** **CYP$PUT_NEXT_BINARY (binary_file, pointer_to_source, file_key, status)**

**Parameters** **binary_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_source**: ^SEQ ( * );

The data to be written.

**file_key**: VAR of integer;

Returns the file cell address at which the write started.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- The data is written to the current position of the file.
- The end-of-information for a binary file follows the last physical cell written to the file. Thus, the file can be written, repositioned backwards, and written again without affecting the end-of-information.
- The size of the data block written to a binary file is determined by the POINTER_TO_SOURCE parameter. CYBIL I/O does not perform any blocking of data. Thus, writing varying length blocks of data at random file addresses can cause previously written data blocks to be partially or fully overwritten.
- Attempting to use this procedure on a file not opened as a binary-type file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write access or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$PUT_KEYED_BINARY

**Purpose** Writes data to a binary file.

**Format** **CYP$PUT_KEYED_BINARY (binary_file, pointer_to_source, file_key, status)**

**Parameters** **binary_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_source**: ^SEQ ( * );

The data to be written.

**file_key**: integer;

The file cell address at which the write is to begin.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- The size of the data block written to a binary file is determined by the POINTER_TO_SOURCE parameter. CYBIL I/O does not perform any blocking of data. Thus, writing varying length blocks of data at random file addresses can cause previously written data blocks to be partially or fully overwritten.
- Attempting to use this procedure on a file not opened as a binary-type file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write access or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## Binary File Positioning

Binary files can be positioned to beginning or end-of-information (with the CYP$POSITION_FILE_AT_BEGINNING and CYP$POSITION_FILE_AT_END procedures described in chapter 11). They can also be positioned to any random file address within the bounds of the file. This is done with CYP$POSITION_BINARY_AT_KEY and CYP$BINARY_FILE_KEY, which are described on the following pages.

## CYP$POSITION_BINARY_AT_KEY

**Purpose** Positions a binary file to a specified file cell address.

**Format** **CYP$POSITION_BINARY_AT_KEY (binary_file, file_key, status)**

**Parameters** **binary_file**: cyt$file;

File identifier established when the file was opened.

**file_key**: integer;

The file cell address to which the file is to be positioned.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_operation
cye$key_past_eoi

**Remarks**

- If the FILE_KEY parameter specifies a cell beyond the end-of-information, CYP$GET_KEYED_BINARY will return CYE$KEY_PAST_EOI in the status variable and the position of the file remains unchanged.
- Attempting to use this procedure on a file not opened as a binary-type file will return CYE$INCORRECT_OPERATION in the status variable.

## CYP$BINARY_FILE_KEY Function

**Purpose** Returns the file cell address at which a binary file is currently positioned.

**Format** **CYP$BINARY_FILE_KEY (binary_file)**: integer;

**Parameters** **binary_file**: cyt$file;

File identifier established when the file was opened.

**Remarks**

- If this function is immediately preceeded by a get or put procedure call, the value returned points to the last cell transferred + 1. If this call is immediately preceeded by a CYP$POSITION_BINARY_AT_KEY call, the value returned is the file cell address to which the file was positioned.
- Attempting to use this function on a file not opened as a binary-type file will return a meaningless result.

## Program Examples Using Binary Files

This section contains CYBIL programs which employ binary file procedures.

These programs perform random access on binary files. In the first example, a library of text modules is created from a text file. The modules on the source (text) file are represented as a list of lines whose first line contains the module name only. The module is terminated by an end-of-block, end-of-partition, or an end-of-information.

The second example extracts one of the modules from this library and copies it to a file whose name is that of the module.

Example 1: Create Text Library

```
MODULE create_text_library;

*copyc cyp$open_file
*copyc cyp$get_next_binary
*copyc cyp$get_next_line
*copyc cyp$write_end_of_partition
*copyc cyp$current_file_position
*copyc cyp$put_next_binary
*copyc cyp$close_file
*copyc cyp$position_file_at_beginning
*copyc cyp$put_keyed_binary


  TYPE
    directory_descriptor = record
      key: integer,
      length: integer,
    recend,

    directory_entry = record
      name: string (7),
      length: integer,
      key: integer,
    recend;

  CONST
    source_name = 'SOURCE',
    lib_name = 'LIBRARY',
    directory_name = 'SCRATCH';
```

```
PROGRAM create;

  VAR
    source_file_specs: [STATIC] array [1 .. 4] of
      cyt$file_specification := [[cyc$file_access,
      cyc$read], [cyc$file_kind, cyc$text_file],
      [cyc$file_existence, cyc$old_file],
      [cyc$open_position, cyc$beginning]],
    directory_file_specs: [STATIC] array [1 .. 3] of
      cyt$file_specification := [[cyc$file_kind,
      cyc$binary_file], [cyc$open_position,
      cyc$beginning], [cyc$close_file_disposition,
      cyc$return_file]],
    library_file_specs: [STATIC] array [1 .. 3] of
      cyt$file_specification := [[cyc$file_access,
      cyc$write], [cyc$file_kind, cyc$binary_file],
      [cyc$open_position, cyc$beginning]],
    source_file: cyt$file,
    library_file: cyt$file,
    directory_file: cyt$file,
    directory: directory_descriptor,
    current_module: directory_entry,
    line: string (256),
    line_length: integer,
    module_index: integer,
    first_key: integer,
    dummy_key: integer,
    cells_read: integer,
    read_status: ost$status,
    write_status: ost$status,
    status: ost$status;

  PROCEDURE copy_a_module (VAR module_status: ost$status);

    VAR
      copy_status: ost$status,
      get_status: ost$status,
      put_status: ost$status;


    PROCEDURE copy_the_module_text (VAR local_status:
      ost$status);
    VAR
      get_status: ost$status,
      put_status: ost$status;

    local_status.normal := TRUE;

  /copy_text_loop/
    WHILE TRUE DO
      cyp$get_next_line (source_file, line, line_length,
        get_status);
      IF NOT get_status.normal THEN
        EXIT /copy_text_loop/;
      IFEND;

      CASE cyp$current_file_position (source_file) OF
      = cyc$end_of_information, cyc$end_of_partition,
      cyc$end_of_block =
      ELSE
        current_module.length := current_module.length + 1;
        cyp$put_next_binary (library_file, #SEQ (line_length),
          dummy_key, put_status);
        IF put_status.normal THEN
          cyp$put_next_binary (library_file, #SEQ (line (1,
            line_length)), dummy_key, put_status);
        IFEND;
        IF NOT put_status.normal THEN
          EXIT /copy_text_loop/;
        IFEND;
      CASEND;
    WHILEND /copy_text_loop/;

    local_status.normal := get_status.normal AND
      put_status.normal;

  PROCEND copy_the_module_text;

/copy_module_loop/
  WHILE TRUE DO
    cyp$get_next_line (source_file, line, line_length,
      get_status);
    IF NOT get_status.normal THEN
      EXIT /copy_module_loop/;
    IFEND;
```

```
          CASE cyp$current_file_position (source_file) OF
          = cyc$end_of_information, cyc$end_of_partition,
          cyc$end_of_block = EXIT /copy_module_loop/;
          ELSE
            directory.length := directory.length + 1;
            current_module.name := line (1, line_length);
            current_module.length := 1;
            cyp$put_next_binary (library_file,
              #SEQ (current_module.name), current_module.key,
              put_status);
            IF NOT put_status.normal THEN
              EXIT /copy_module_loop/;
            IFEND;

            copy_the_module_text (copy_status);
            IF NOT copy_status.normal THEN
              EXIT /copy_module_loop/;
            IFEND;

            cyp$put_next_binary (directory_file,
              #SEQ (current_module), dummy_key, put_status);
            IF NOT put_status.normal THEN
              EXIT /copy_module_loop/;
            IFEND;
          CASEND;
        WHILEND /copy_module_loop/;
        module_status.normal := copy_status.normal AND
          put_status.normal AND get_status.normal;

      PROCEND copy_a_module;

      PROCEDURE copy_directory_to_library (VAR local_status:
        ost$status);

        VAR
          module_index: integer,
          read_status: ost$status,
          write_status: ost$status;

        cyp$get_next_binary (directory_file, #SEQ (current_module),
          dummy_key, cells_read, read_status);
        IF read_status.normal THEN
          cyp$put_next_binary (library_file, #SEQ (current_module),
                directory.key, write_status);
          IF write_status.normal THEN
          /read_loop/
            FOR module_index := 2 TO directory.length DO
              cyp$get_next_binary (directory_file,
                #SEQ (current_module), dummy_key, cells_read,
                read_status);
              IF NOT read_status.normal THEN
                EXIT /read_loop/;
              IFEND;
              cyp$put_next_binary (library_file,
                #SEQ (current_module), dummy_key, write_status);
              IF NOT write_status.normal THEN
                EXIT /read_loop/;
              IFEND;
            FOREND /read_loop/;
            IF read_status.normal AND write_status.normal THEN
              cyp$put_keyed_binary (library_file, #SEQ (directory),
              first_key, write_status);
            IFEND;
          IFEND;
        IFEND;

        local_status.normal := read_status.normal AND
          write_status.normal;

      PROCEND copy_directory_to_library;


      cyp$open_file (source_name, ^source_file_specs, source_file,
        status);
      IF status.normal THEN
        cyp$open_file (directory_name, ^directory_file_specs,
          directory_file, status);
        IF status.normal THEN
          cyp$open_file (lib_name, ^library_file_specs,
            library_file, status);
        IFEND;
      IFEND;

      IF status.normal THEN

      /main_program/
        BEGIN
  {*}
  {  reserve space for a directory
```

```
{*}
        directory.length := 0;
        cyp$put_next_binary (library_file, #SEQ (directory),
          first_key, write_status);
        IF write_status.normal THEN
          copy_a_module (read_status);
          cyp$close_file (source_file, cyc$end, status);
          IF ((read_status.normal) AND (directory.length > 0))
            THEN
              cyp$position_file_at_beginning (directory_file,
              status);
            IF NOT status.normal THEN
              EXIT /main_program/;
            IFEND;

            copy_directory_to_library (status);

          IFEND;
        IFEND;

      END /main_program/;

    IFEND;
    cyp$close_file (directory_file, cyc$asis, status);
    cyp$close_file (library_file, cyc$beginning, status);

  PROCEND create;

MODEND create_text_library;
```

## Example 2: Extract From Text Library

This example extracts one of the modules from the library created in the first example and copies it to a file whose name is that of the module.

```
MODULE extract_from_text_library;

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$get_next_binary
*copyc cyp$get_keyed_binary
*copyc cyp$position_binary_at_key
*copyc cyp$put_keyed_binary
*copyc cyp$put_next_line
*copyc cyp$current_file_position

  TYPE
    directory_descriptor = record
      key: integer,
      length: integer,
    recend,

    directory_entry = record
      name: string (7),
      length: integer,
      key: integer,
    recend;

  CONST
    lib_name = 'LIBRARY';

  CONST
    name_of_module = 'TEXTMOD';


  PROGRAM extract;

    VAR
      library_file_specs: [STATIC] array [1 .. 4] of
        cyt$file_specification := [[cyc$file_kind,
        cyc$binary_file], [cyc$open_position, cyc$beginning],
        [cyc$file_existence, cyc$old_file], [cyc$file_access,
        cyc$read]],
      output_file_specs: [STATIC] array [1 .. 3] of
        cyt$file_specification := [[cyc$file_access, cyc$write],
        [cyc$file_kind, cyc$text_file], [cyc$open_position,
        cyc$beginning]],
      library_file: cyt$file,
      out_file: cyt$file,
      directory: directory_descriptor,
      current_module: directory_entry,
      line: string (256),
      line_length: integer,
      module_found: boolean,
      dummy_key: integer,
      cells_read: integer,
      status: ost$status;

    PROCEDURE search_for_module (library_directory:
      directory_descriptor;
      VAR module_is_in_directory: boolean;
      VAR search_status: ost$status);

      VAR
        module_index: integer;


      module_is_in_directory := FALSE;
      search_status.normal := TRUE;

      cyp$position_binary_at_key (library_file,
        library_directory.key, search_status);
      IF NOT search_status.normal THEN
        RETURN; {----->
      IFEND;

    /search_directory/
      FOR module_index := 1 TO library_directory.length DO
        cyp$get_next_binary (library_file, #SEQ (current_module),
          dummy_key, cells_read, search_status);
        IF NOT search_status.normal THEN
          RETURN; {----->
        IFEND;
        IF current_module.name = name_of_module THEN
          module_is_in_directory := TRUE;
          EXIT /search_directory/;
        IFEND;
      FOREND /search_directory/;

    PROCEND search_for_module;
```

```
PROCEDURE copy_the_module_text (VAR copy_status: ost$status);

/module_loop/
  WHILE current_module.length > 1 DO
    cyp$get_next_binary (library_file, #SEQ (line_length),
      dummy_key, cells_read, copy_status);
    IF NOT copy_status.normal THEN
      EXIT /module_loop/;
    IFEND;
    cyp$get_next_binary (library_file, #SEQ (line (1,
      line_length)), dummy_key, cells_read, copy_status);
    IF NOT copy_status.normal THEN
      EXIT /module_loop/;
    IFEND;
    cyp$put_next_line (out_file, line (1, line_length),
      copy_status);
    IF NOT copy_status.normal THEN
      EXIT /module_loop/;
    IFEND;
    current_module.length := current_module.length - 1;
  WHILEND /module_loop/;
PROCEND copy_the_module_text;

cyp$open_file (lib_name, ^library_file_specs, library_file,
  status);
IF NOT status.normal THEN
  RETURN; {----->
IFEND;
cyp$get_next_binary (library_file, #SEQ (directory),
  dummy_key, cells_read, status);
IF NOT status.normal THEN
  RETURN; {----->
IFEND;
IF directory.length = 0 THEN
  RETURN; {----->
IFEND;
search_for_module (directory, module_found, status);

IF status.normal AND module_found THEN
  cyp$open_file (name_of_module, ^output_file_specs,
    out_file, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
```

```
        cyp$get_keyed_binary (library_file,
          #SEQ (current_module.name), current_module.key,
          cells_read, status);
        IF NOT status.normal THEN
          RETURN; {----->
        IFEND;
        cyp$put_next_line (out_file, current_module.name, status);
        IF NOT status.normal THEN
          RETURN; {----->
        IFEND;

        copy_the_module_text (status);
      IFEND;
      cyp$close_file (library_file, cyc$beginning, status);
      IF NOT status.normal THEN
        RETURN; {----->
      IFEND;

      cyp$close_file (out_file, cyc$beginning, status);
      IF NOT status.normal THEN
        RETURN; {----->
      IFEND;

    PROCEND extract;

  MODEND extract_from_text_library;
```

## Record Files

The procedures and functions described in this section are for use with record files only. To use other types of files with CYBIL I/O, refer to the appropriate section of this chapter.

The data transfer procedures for record files (like any programmer-defined procedures in CYBIL) must have parameters of a specific CYBIL type. To transfer data to or from a record file, the CYBIL type of the parameter that specifies the data to be read or written must match the CYBIL type of the program variable that contains the data to be read or written. The CYBIL I/O procedures that read and write on record files require that the data be specified as a pointer to a CYBIL sequence. Programs using the record file procedures must therefore specify the data as a variable of type pointer to CYBIL sequence. This pointer is usually defined by using the CYBIL #SEQ function.

For example, given the following CYBIL variable declarations:

```
VAR
  data_item_1: my_data_type,
  data_item_2: ^my_data_type,
  data_item_3: ^array [1 .. 50] of my_data_type;
```

the pointers to CYBIL sequences may be defined as follows:

```
  #SEQ (data_item_1)
  #SEQ (data_item_2^)
  #SEQ (data_item_3^)
  #SEQ (data_item_3^ [5])
```

For examples using the CYBIL #SEQ function to pass data to or from the record-file read/write procedures, refer to Program Examples Using Record Files later in this chapter.

Data is read from or written to record type files as full or partial records. These records are not to be confused with the CYBIL record type.

### Record File Structure

In record files, data exists as a sequence of logical records each of which is terminated with an end-of-record. CYBIL I/O allows the reading and writing of both full and partial records. That is, a record may be transferred as the result of a single read or write operation, or a record may be transferred as the result of several partial read or write operations. Record file reads and writes map the data to a CYBIL data structure. For example, a CYBIL array may be written as a record or partial record. The address and size of the data structure are passed to CYBIL I/O as a CYBIL sequence pointer. CYBIL I/O uses this information to write a record that exactly corresponds byte for byte with the way the data is stored in the CYBIL data structure.

CYBIL I/O supports only sequential access of record files. Data appears on such files in the order in which it was written, and can only be read in the same order.

Record files may be positioned to the beginning-of-information or end-of-information. In addition, record files may be positioned forward or backward a user-specified number of records or partitions. Positioning a record file backwards and then writing to the file means that any data following the data just written to the file is lost.

The end-of-information always immediately follows the last data written to the file.

The following procedures and functions may be used with record files only. These procedures and functions are described in greater detail on the following pages.

CYP$GET_NEXT_RECORD

Reads the next record from a record file.

CYP$GET_PARTIAL_RECORD

Reads a portion of a record from a record file.

CYP$PUT_NEXT_RECORD

Writes a record on a record file.

CYP$PUT_PARTIAL_RECORD

Writes a partial record on a record file.

CYP$POSITION_RECORD_FILE

Allows a record file to be repositioned.

## CYP$GET_NEXT_RECORD

**Purpose** Reads the next record from a record file.

**Format** **CYP$GET_NEXT_RECORD (record_file, pointer_to_target, number_of_cells_read, status)**

**Parameters** **record_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_target**: ^SEQ ( * );

The data structure into which data is to be read.

**number_of_cells_read**: VAR of integer;

Returns the number of cells actually read.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation

**Remarks**

- If the current file position is not at the beginning of a record, the file is positioned forward to the beginning of the next record or partition before the read begins.

- CYBIL I/O reads data from the file until it encounters the end-of-record or the end of the data structure specified by the POINTER_TO_TARGET parameter. The NUMBER_OF_CELLS_READ parameter will return the number of data cells actually read into the data structure specified by POINTER_TO_TARGET.

- If the read terminates because the end-of-record was encountered, the CYP$CURRENT_FILE_POSITION function will return CYC$END_OF_RECORD. If the read terminates because CYBIL I/O encountered the end of the POINTER_TO_TARGET data structure, the CYP$CURRENT_FILE_POSTIION function will return CYC$MIDDLE_OF_RECORD. To read the remainder of the record, the program must issue CYP$GET_

PARTIAL_RECORD calls until the CYP$CURRENT_FILE_POSITION function returns a value of CYC$END_OF_RECORD. (The CYP$CURRENT_FILE_POSITION function is described in chapter 11.)

- If an end-of-partition is encountered, no data is read, the NUMBER_OF_CELLS_READ parameter returns a value of 0 (zero), and the CYP$CURRENT_FILE_POSITION function will return a value of CYC$END_OF_PARTITION.

- If an end-of-information is encountered, no data is read, the NUMBER_OF_CELLS_READ parameter returns a value of 0 (zero), and the CYP$CURRENT_FILE_POSITION function will return a value of CYC$END_OF_INFORMATION.

- Attempting to use this procedure on a file not opened as a record file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

## CYP$GET_PARTIAL_RECORD

**Purpose** Reads a portion of a record from a record file.

**Format** **CYP$GET_PARTIAL_RECORD (record_file, pointer_to_target, number_of_cells_read, last_part_of_record, status)**

**Parameters** **record_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_target**: ^SEQ ( * );

Specifies the data structure into which data is to be read.

**number_of_cells_read**: VAR of integer;

Returns the number of cells actually read.

**last_part_of_record**: VAR of boolean;

Returns a value of TRUE if the end-of-record was encountered, and a value of FALSE otherwise.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation

**Remarks**

- Reading begins at the current position of the file and continues until the end-of-record or the end of the data structure specified by POINTER_TO_TARGET is encountered. The NUMBER_OF_CELLS_READ parameter will return the number of data cells actually read into the data structure specified by POINTER_TO_TARGET.

- If the read terminates because the end of the record was encountered, the CYP$CURRENT_FILE_POSITION function will return CYC$END_OF_RECORD. If the read terminates because the end of the POINTER_TO_TARGET data structure was encountered, the CYP$CURRENT_FILE_POSITION function will return CYC$MIDDLE_OF_RECORD. To read the remainder of the record, the program must issue CYP$GET_PARTIAL_RECORD calls until the CYP$CURRENT_FILE_POSITION function returns a value of CYC$END_OF_RECORD.

- If the end-of-partition is encountered, no data is read, the NUMBER_OF_CELLS_READ parameter returns a value of 0 (zero), and the CYP$CURRENT_FILE_POSITION function will return a value of CYC$END_OF_PARTITION.

- If the end-of-information is encountered, no data is read, the NUMBER_OF_CELLS_READ parameter returns a value of 0 (zero), and the CYP$CURRENT_FILE_POSITION function will return a value of CYC$END_OF_INFORMATION.

- Attempting to use this procedure on a file not opened as a record file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

## CYP$PUT_NEXT_RECORD

**Purpose** Writes a record on a record file.

**Format** **CYP$PUT_NEXT_RECORD (record_file, pointer_to_source, status)**

**Parameters** **record_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_source**: ^SEQ ( * );

The data to be written. The data is written as a complete record. If the last write to the file was made with CYP$PUT_PARTIAL_RECORD, that record is completed before the data specified by this parameter is written as a new complete record.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- The end-of-information on a record file immediately follows the data last written. Thus, if you write to a record file, and then position the file to its beginning (or perform a backward record skip) and again write to the file, data will be lost.

- Attempting to use this procedure on a file not opened as a record file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file not opened with either write access or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$PUT_PARTIAL_RECORD

**Purpose** Writes a partial record on a record file.

**Format** **CYP$PUT_PARTIAL_RECORD (record_file, pointer_to_source, last_part_of_record, status)**

**Parameters** **record_file**: cyt$file;

File identifier established when the file was opened.

**pointer_to_source**: ^SEQ ( * );

Specifies the data to be written.

**last_part_of_record**: boolean;

Specifies whether or not more data can be appended to the current record. If this parameter is TRUE, the data specified by the POINTER_TO_SOURCE parameter is written to the file and the record is terminated. The next full or partial write to the file will begin a new record.

If this parameter is FALSE, the data specified by the POINTER_TO_SOURCE parameter is written to the file but the record is not terminated. Additional data can be appended to the record if the next write to the file is done by the CYP$PUT_PARTIAL_RECORD procedure.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- The end-of-information on a record file immediately follows the data last written. Thus, if you write to a record file, and then position the file to its beginning (or perform a backward record skip) and again write to the file, data will be lost.

- Attempting to use this procedure on a file not opened as a record file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file not opened with either write access or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## Record File Positioning

On NOS/VE, record files can be subdivided into records or partitions. They can be positioned to either beginning or end-of-information (with the CYP$POSITION_FILE_AT_BEGINNING and CYP$POSITION_FILE_AT_END procedures, described in chapter 11). Positioning may only be performed on record files that were opened for read or read write access.

Record files can also be positioned forward or backward one or more records or partitions with the CYP$POSITION_RECORD_FILE procedure, described on the following pages.

## CYP$POSITION_RECORD_FILE

**Purpose** Repositions a record file.

**Format** **CYP$POSITION_RECORD_FILE (record_file, direction, count, unit, status)**

**Parameters** **record_file**: cyt$file;

File identifier established when the file was opened.

**direction**: cyt$skip_direction;

Specifies forward or backward positioning. Enter one of the following values:

CYC$FORWARD

CYC$BACKWARD

**count**: integer;

The number of units the file is to be positioned.

**unit**: cyt$skip_unit;

How the file is to be repositioned (by records or partitions). Enter one of the following values:

CYC$RECORD

CYC$PARTITION

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation
cye$incorrect_skip_count
cye$premature_end_of_operation

**Remarks**

- Attempting to use this procedure on a file not opened as a record file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

- The position of the file after a positioning operation depends on the positioning unit (records or partitions), the initial file position, the number of units positioned, and the positioning direction. Table 12-1 lists positioning results assuming that no boundary condition is detected before the positioning count is exhausted.

**Table 12-1. Results of CYP$POSITION_RECORD_FILE**

| File position before the operation | Positioning operation | Result |
|---|---|---|
| **Positioning by records:** | | |
| cyc$beginning_of_information, end_of_record, end_of_partition, end_of_information | Position forward or backward zero records. | No movement; the file remains the same as before the positioning operation. |
| middle_of_record | Position forward zero records. | The file is positioned to the end of the current record. |
| middle_of_record | Position backward zero records. | The file is positioned to the end of the preceeding record. |
| End of record N | Position forward one or more (M) records. | The file is positioned to the end of record N + M. |
| End of record N | Position backward one or more (M). records. | The file is positioned to the end of record N - M. |
| **Positioning by partitions:** | | |
| beginning_of_information, end_of_information | Position forward or backward zero partitions. | No movement; the file remains positioned the same as before the positioning operation. |

| | | |
|---|---|---|
| middle_of_record, end_of_record, end_of_partition | Position forward zero partitions. | The file is positioned to the beginning of the next partition. |
| middle_of_record, end_of_record, end_of_partition | Position backward zero partitions. | The file is positioned to the beginning of the current partition. |
| middle_of_record, end_of_record, end_of_partition | Position forward one or more (M) partitions. | The file is positioned to the beginning of partition (current + M +1). |
| middle_of_record, end_of_record, end_of_partition | Position backward one or more (M) partitions. | The file is positioned to the beginning of partition (current - M). |

In Table 12-1 it is assumed that no boundary conditions are encountered during the positioning operation. If CYP$POSITION_RECORD_FILE encounters a boundary condition before the count is exhausted, the positioning operation stops at the boundary and CYE$PREMATURE_END_OF_OPERATION will be returned in the status variable.

The following are the boundary conditions:

- A position forward by records encounters an end-of-partition or end-of-information.
- A position forward by partitions encounters end-of-information.
- A position backwards by records encounters an end-of-partition or beginning-of-information.
- A position backwards by partitions encounters beginning-of-information.

## Program Example Using Record Files

The following example illustrates the use of record file procedures. The input file is assumed to contain several kinds of logical records. An id-record identifies the record following it as either an employee record or a vendor record. A vendor record is followed by one or more product records. This program produces a list of vendor names and the names of the products supplied by each vendor.

Example 1: Extract Information From Records

```
MODULE list_vendor_and_products;

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$get_next_record
*copyc cyp$put_next_line
*copyc cyp$put_partial_line
*copyc cyp$position_record_file
*copyc cyp$tab_file
*copyc cyp$current_file_position


  PROGRAM list_vendor_and_products;

    CONST
      in_name = 'EMPDB',
      out_name = 'EMPLIST';

    TYPE
      full_name = record
        first: string (10),
        initial: char,
        last: string (15),
      recend,

      employee_entry = record
        number: 0 .. 999999,
        name: full_name,
        department_number: 0 .. 9999,
        department_name: string (20),
      recend,

      vendor_entry = record
        number: 0 .. 99999999,
        name: string (30),
      street_address: string (30),
      city_state: string (30),
      zip_code: 0 .. 99999,
      number_of_products: integer,
    recend,

    product_entry = record
      name: string (20),
      product_number: string (10),
    recend,

    entry_id = (employee_id, vendor_id);

  VAR
    in_file: cyt$file,
    out_file: cyt$file,
    in_file_specs: cyt$file_specifications,
    out_file_specs: cyt$file_specifications,
    cells_read: integer,
    vendor: vendor_entry,
    product: product_entry,
    record_id: entry_id,
    i: integer,
    status: ost$status;

  PUSH in_file_specs: [1 .. 4];
  in_file_specs^ [1].selector := cyc$file_kind;
  in_file_specs^ [1].file_kind := cyc$record_file;
  in_file_specs^ [2].selector := cyc$file_access;
  in_file_specs^ [2].file_access := cyc$read;
  in_file_specs^ [3].selector := cyc$file_existence;
  in_file_specs^ [3].file_existence := cyc$old_file;
  in_file_specs^ [4].selector := cyc$open_position;
  in_file_specs^ [4].open_position := cyc$beginning;

  PUSH out_file_specs: [1 .. 3];
  out_file_specs^ [1].selector := cyc$file_kind;
  out_file_specs^ [1].file_kind := cyc$text_file;
  out_file_specs^ [2].selector := cyc$file_access;
  out_file_specs^ [2].file_access := cyc$write;
  out_file_specs^ [3].selector := cyc$open_position;
  out_file_specs^ [3].open_position := cyc$beginning;


  cyp$open_file (in_name, in_file_specs, in_file, status);
```

```
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    WHILE status.normal DO
      cyp$get_next_record (in_file, #SEQ (record_id),
        cells_read, status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
      CASE cyp$current_file_position (in_file) OF
      = cyc$end_of_partition, cyc$end_of_block =
        CYCLE /main_loop/;
      = cyc$end_of_information =
        EXIT /main_loop/;
      = cyc$middle_of_record =
        cyp$put_next_line (out_file, 'ERROR reading input file',
          status);
        EXIT /main_loop/;
      = cyc$end_of_record =
        CASE record_id OF
        = employee_id =
          cyp$position_record_file (in_file, cyc$forward, 1,
            cyc$record, status);
          IF NOT status.normal THEN
            EXIT /main_loop/;
          IFEND;
        = vendor_id =
          cyp$get_next_record (in_file, #SEQ (vendor),
            cells_read, status);
          IF NOT status.normal THEN
            EXIT /main_loop/;
          IFEND;
          IF cyp$current_file_position (in_file) =
            cyc$end_of_record THEN
               cyp$put_next_line (out_file, vendor.name,
                 status);
            IF NOT status.normal THEN
              EXIT /main_loop/;
            IFEND;
          FOR i := 1 TO vendor.number_of_products DO
            cyp$get_next_record (in_file, #SEQ (product),
              cells_read, status);
            IF cyp$current_file_position (in_file) <>
              cyc$end_of_record THEN
                 cyp$put_next_line (out_file, 'ERROR reading
                 input file', status);
              EXIT /main_loop/;
            ELSEIF (NOT status.normal) OR (cells_read <>
              #SIZE (product)) THEN
                EXIT /main_loop/;
            IFEND;
            cyp$tab_file (out_file, 10, status);
            IF NOT status.normal THEN
              EXIT /main_loop/;
            IFEND;
            cyp$put_partial_line (out_file, product.name,
              TRUE, status);
            IF NOT status.normal THEN
              EXIT /main_loop/;
            IFEND;
          FOREND;
        ELSE
          cyp$put_next_line (out_file, 'ERROR reading input
            file', status);
          EXIT /main_loop/;
        IFEND;
      CASEND;
    CASEND;
  WHILEND /main_loop/;

  cyp$close_file (in_file, cyc$beginning, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$close_file (out_file, cyc$beginning, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;

PROCEND list_vendor_and_products;

MODEND list_vendor_and_products;
```

## Reading and Writing Text and Display Files

The procedures and functions described in this section are for use with text and display-type files only (they apply to both).

Data is transferred to and from text files and display files as lines or partial lines. Internally, these lines are represented as CYBIL strings of characters. Externally (on the file), lines may be represented in 8-bit ASCII.

The external character set is specified on the FILE_SPECIFICATIONS parameter of the CYP$OPEN_FILE procedure when the file is opened. Data transfers on text or display files may involve a translation between character sets (unlike binary and record file transfers, in which the file data is not modified).

The maximum line-length written to text or display files, and the page size for display files, are specified with the FILE_SPECIFICATIONS parameter on the CYP$OPEN_FILE procedure.

### Text File Structure

A text file, which is a variation of a record file, is assumed to contain character data. Since character data is generally conceived of as lines, text-file records are treated as lines and the end-of-record for text files as end-of-line.

The basic entity on a text file is a line which can be transferred to or from the file in whole or in part. In addition, it is possible to tab to a specified column in an output line and skip a specified number of lines. Text files may be positioned to the beginning-of-information or to the end-of-information.

Data is passed to and from the text file procedures as CYBIL strings rather than as CYBIL sequence pointers. Like record files, text files can only be accessed sequentially.

### Display File Structure

A display file is a special form of write-only text file. Display files should be used when the file is to be printed, or routed to any device which uses format control characters. Format control characters are automatically prefixed to each line written to display-type files.

Display files have additional facilities for vertical format control (described on the following pages). It is possible to limit the number of printed lines on a page, insert a given number of empty lines, overprint lines, or position the next line at a specified line number or at the top of the next display page. Several functions are provided to interrogate certain items of display page information for display files.

Display files may only be written. If it is necessary to read a file which was written as a display file, the file should be accessed as a text file.

With each display file, you may associate a procedure to be called when a page overflow condition occurs for that file. The procedure may be one of your own or a special internal CYBIL I/O procedure that produces a standard title line. (A page-overflow procedure of your own is specified on the FILE_SPECIFICATIONS parameter of CYP$OPEN_FILE. For more information on page-overflow handling, refer to Page-Overflow Processing for Display Files later in this chapter.)

The following procedures and functions may be used with text and display files only. These procedures and functions are described in greater detail on the following pages.

CYP$GET_NEXT_LINE

Reads the next complete line from a text or display file.

CYP$GET_PARTIAL_LINE

Reads a character string from a text or display file.

CYP$PUT_NEXT_LINE

Writes a string of characters to a text or display file.

CYP$PUT_PARTIAL_LINE

Writes a string of characters to a text or display file.

CYP$WRITE_END_OF_LINE

Writes an end-of-line to a text or display file.

CYP$FLUSH_LINE

Flushes the line buffer for a text or display file.

CYP$TAB_FILE

Positions a text or display file to a specified column or position within a line.

CYP$SKIP_LINES

Writes one or more blank lines to a text or display file.

CYP$FILE_CONNECTED_TO_TERMINAL

Determines whether or not a text or display file is connected to a terminal.

CYP$CURRENT_COLUMN

Returns the current column within the current line of a text or display file.

CYP$PAGE_WIDTH

Returns the page width associated with a text or display file.

## CYP$GET_NEXT_LINE

**Purpose** Reads the next complete line from a text or display file.

**Format** **CYP$GET_NEXT_LINE (file, line, number_of_characters_read, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**line**: VAR of string ( * <= cyc$max_page_width);

The CYBIL string into which the line was read. If the line from the file is too long to fit into this string, the line is truncated by skipping to the end of the line after the transfer is complete.

**number_of_characters_read**: VAR of integer;

Returns the number of characters transferred into the string specified by the LINE parameter.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation

**Remarks**

- If the previous transfer was partial, a skip to the end of that line is performed prior to this read.
- A line containing zero characters is returned as an empty string (the NUMBER_OF_CHARACTERS_READ parameter returns a value of zero). A line with no characters is one in which a carriage return was entered in the first position of the line, or is any empty line that was written via a call to CYP$WRITE_END_OF_LINE.
- Attempting to use this procedure on a file not opened as a text or display-type file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

## CYP$GET_PARTIAL_LINE

**Purpose** Reads a character string from a text or display file.

**Format** **CYP$GET_PARTIAL_LINE (file, partial_line, number_of_characters_read, last_part_of_line, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**partial_line**: VAR of string ( * <= cyc$max_page_width);

The CYBIL string into which the character string is read.

**number_of_characters_read**: VAR of integer;

Returns the number of characters transferred into the string specified by the partial_line parameter.

**last_part_of_line**: VAR of boolean;

Returns a value of TRUE if the end of the line was encountered, and a value of FALSE otherwise.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_input_request
cye$incorrect_operation

**Remarks**

- A line containing zero characters is returned as an empty string (the NUMBER_OF_CHARACTERS_READ parameter returns a value of zero). A line with no characters is one in which a carriage return was entered in the first position of the line, or is any empty line that was written via a call to CYP$WRITE_END_OF_LINE.

- Attempting to use this procedure on a file not opened as a text or display-type file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file opened for write access will return CYE$INCORRECT_INPUT_REQUEST in the status variable.

## CYP$PUT_NEXT_LINE

**Purpose** Writes a string of characters to a text or display file.

**Format** **CYP$PUT_NEXT_LINE (file, line, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**line**: string ( * <= cyc$max_page_width);

The string of characters to be written as a complete line. If the last write to the file was a partial line, that line is first completed, and then the characters specified by this parameter are written.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the length of the character string exceeds the page width, the line will be truncated.

- For a display file, format control characters are automatically prefixed to the new line. (The format control characters that are used depend on what kind of write operation preceded this call to CYP$PUT_NEXT_LINE.) In addition, if displaying the line causes the display page length to be exceeded, the page overflow mechanism is executed.

- Attempting to use this procedure on a file not opened as a text-type or display-type file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file not opened for write access or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$PUT_PARTIAL_LINE

**Purpose** Writes a string of characters to a text or display file.

**Format** **CYP$PUT_PARTIAL_LINE (file, partial_line, last_part_of_line, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**partial_line**: string ( * <= cyc$max_page_width);

The string of characters to be written.

**last_part_of_line**: boolean;

Whether or not more characters can be written to the current line. If the LAST_PART_OF_LINE parameter is TRUE, an end-of-line is appended to the current line after the character string is written. If the LAST_PART_OF_LINE parameter is FALSE, subsequent CYP$PUT_PARTIAL_LINE calls may append data to the current line.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- For a display file, format control characters are automatically prefixed to the beginning of each new line. (The format control characters that are used depend on what kind of write operation preceded this call to CYP$PUT_PARTIAL_LINE.) In addition, if the LAST_PART_OF_LINE parameter is TRUE and displaying the current line causes the display page length to be exceeded, the page overflow mechanism is executed.
- If the length of the current line exceeds the page width, the line will be truncated.

- Attempting to use this procedure on a file not opened as a text or display file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$WRITE_END_OF_LINE

**Purpose** Writes an end-of-line to a text or display file.

**Format** **CYP$WRITE_END_OF_LINE (file, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the last write to the file was partial, that line is completed; otherwise, an empty line results.
- Attempting to use this procedure on a file not opened as a text or display file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$FLUSH_LINE

**Purpose** Flushes the line buffer for a text or display file.

**Format** **CYP$FLUSH_LINE (file, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the line buffer contains data, the line is terminated and then written to the specified file. If the line buffer contains no data, this procedure results in no operation on the file.

- Attempting to use this procedure on a file not opened as a text or display file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$TAB_FILE

**Purpose** Positions a text or display file to a specified column or position within a line.

**Format** **CYP$TAB_FILE (file, tab_column, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**tab_column**: cyt$page_width;

The column to which the file should be positioned.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- This procedure performs a write to the file.
- If the TAB_COLUMN parameter is less than or equal to the file's current column, this procedure does nothing. Otherwise, sufficient space characters are written to the file so that the next partial write to the file will begin at the column specified by TAB_COLUMN.
- If the TAB_COLUMN parameter is larger than the page width of the device associated with the file, line truncation will occur when the line is written.
- Attempting to use this procedure on a file not opened as a text or display file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$SKIP_LINES

**Purpose** Writes one or more blank lines to a text or display file.

**Format** **CYP$SKIP_LINES (file, number_of_lines, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**number_of_lines**: integer;

The number of blank lines to be written. If the last write to the file was partial, that line is first completed and then the number of blank lines specified by this parameter are written to the file.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the file was opened as a display file and the NUMBER_OF_LINES parameter was specified as -1, the next line written to the file will overwrite the current line. In addition, if the NUMBER_OF_LINES plus the current line number exceeds the display page size, the page overflow mechanism will be executed.

- Attempting to use this procedure on a file not opened as a text or display file will return CYE$INCORRECT_OPERATION in the status variable.

- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$FILE_CONNECTED_TO_TERMINAL Function

**Purpose** Determines whether or not a text or display file is connected to a terminal.

**Format** **CYP$FILE_CONNECTED_TO_TERMINAL (file):** boolean;

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**Remarks**

- Returns a value of TRUE if the file is connected to a terminal. Otherwise, a value of FALSE is returned.
- This function may be used to determine whether the calling program needs to limit line size or perform any special data formatting for terminal files.
- Attempting to use this function on a file not opened as a text or display file will return CYE$INCORRECT_OPERATION in the status variable.

## CYP$CURRENT_COLUMN Function

**Purpose** Returns the current column within the current line of a text or display file.

**Format** **CYP$CURRENT_COLUMN (file)**: cyt$page_width;

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**Remarks**

- This function returns the column at which the next read or write will begin.
- Attempting to use this function on a file not opened as a text or display file will return an undefined result.

## CYP$PAGE_WIDTH Function

**Purpose** Returns the page width associated with a text or display file.

**Format** **CYP$PAGE_WIDTH (file)**: cyt$page_width;

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**Remarks** Attempting to use this function on a file not opened as a text or display file will return an undefined result.

## Program Examples Using Text and Display Files

The following example illustrates the use of text file procedures for copying one text file to another. Only data between selected columns on the old file is written to the new file, and within those columns trailing space characters are deleted.

Example 1: Copy Column Range of Text File

```
MODULE truncate;

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$get_next_line
*copyc cyp$put_next_line
*copyc cyp$write_end_of_line
*copyc cyp$write_end_of_block
*copyc cyp$write_end_of_partition
*copyc cyp$current_file_position


  PROGRAM truncate;

    CONST
      in_name = 'OLD',
      out_name = 'NEW',
      leftmost_column_# = 11,
      rightmost_column_# = 72;

    VAR
      in_file: cyt$file,
      out_file: cyt$file,
      in_file_specs: cyt$file_specifications,
      out_file_specs: cyt$file_specifications,
      line_ptr: ^string ( * <= cyc$max_page_width),
      line_length: integer,
      status: ost$status;

    PUSH in_file_specs: [1 .. 4];
    in_file_specs^ [1].selector := cyc$file_kind;
    in_file_specs^ [1].file_kind := cyc$text_file;
    in_file_specs^ [2].selector := cyc$file_access;
    in_file_specs^ [2].file_access := cyc$read;
    in_file_specs^ [3].selector := cyc$file_existence;
    in_file_specs^ [3].file_existence := cyc$old_file;
    in_file_specs^ [4].selector := cyc$open_position;
    in_file_specs^ [4].open_position := cyc$beginning;

    PUSH out_file_specs: [1 .. 3];
    out_file_specs^ [1].selector := cyc$file_kind;
    out_file_specs^ [1].file_kind := cyc$text_file;
    out_file_specs^ [2].selector := cyc$file_access;
    out_file_specs^ [2].file_access := cyc$write;
    out_file_specs^ [3].selector := cyc$open_position;
    out_file_specs^ [3].open_position := cyc$beginning;

    ALLOCATE line_ptr: [rightmost_column_#];

    cyp$open_file (in_name, in_file_specs, in_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    WHILE status.normal DO
      cyp$get_next_line (in_file, line_ptr^, line_length,
        status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
      CASE cyp$current_file_position (in_file) OF
      = cyc$end_of_partition =
        cyp$write_end_of_partition (out_file, status);
        IF NOT status.normal THEN
          EXIT /main_loop/;
        IFEND;
      = cyc$end_of_block =
        cyp$write_end_of_block (out_file, status);
        IF NOT status.normal THEN
          EXIT /main_loop/;
        IFEND;
      = cyc$end_of_information =
        EXIT /main_loop/;
      ELSE
        WHILE (line_length > leftmost_column_#) AND (line_ptr^
          (line_length) = ' ') DO
             line_length := line_length - 1;
          WHILEND;
          line_length := line_length - leftmost_column_# + 1;
          IF line_length > 0 THEN
            cyp$put_next_line (out_file, line_ptr^
            (leftmost_column_#, line_length), status);
          ELSE
            cyp$write_end_of_line (out_file, status);
          IFEND;
          IF NOT status.normal THEN
            EXIT /main_loop/;
          IFEND;
        CASEND;
      WHILEND /main_loop/;

      cyp$close_file (in_file, cyc$beginning, status);
      IF NOT status.normal THEN
        RETURN; {----->
      IFEND;
      cyp$close_file (out_file, cyc$beginning, status);
      IF NOT status.normal THEN
        RETURN; {----->
      IFEND;

      FREE line_ptr;

    PROCEND truncate;

  MODEND truncate;
```

The following example illustrates the use of display and text file procedures. Note the procedure for processing page-overflow. (The display file procedures are described later in this chapter.)

Example 2: Display a Text File

```
MODULE display_file_example;

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$current_file_position
*copyc cyp$current_display_line
*copyc cyp$get_partial_line
*copyc cyp$display_page_length
*copyc cyp$tab_file
*copyc cyp$skip_lines
*copyc cyp$position_display_page
*copyc cyp$put_partial_line
*copyc cyp$write_end_of_line
*copyc cyp$display_page_eject
*copyc cyp$current_page_number


  CONST
    in_name = 'TEXFILE';

  VAR
    file_numb: integer := 1,
    record_numb: integer := 1;


  PROGRAM list;

    CONST
      out_page_width = 80,
      out_page_length = 50,
      footing_line_number = out_page_length - 2,
      out_name = 'OUTPUT';

    VAR
      in_file_specs: cyt$file_specifications,
      out_file_specs: cyt$file_specifications,
      in_file: cyt$file,
      out_file: cyt$file,
      line: string (80),
      line_length: integer,
    eol: boolean,
    status: ost$status;

  PROCEDURE my_new_page_proc (print_file: cyt$file;
        next_page_number: integer;
    VAR status: ost$status);

    VAR
      str_holder: string (10),
      str_length: integer;

    cyp$display_page_eject (print_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_partial_line (print_file, 'LISTING OF ', FALSE,
      status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_partial_line (print_file, in_name, FALSE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$tab_file (print_file, 50, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_partial_line (print_file, 'FILE ', FALSE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    STRINGREP (str_holder, str_length, file_numb);
    cyp$put_partial_line (print_file, str_holder
      (1, str_length), FALSE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_partial_line (print_file, RECORD ', FALSE, ',
      status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    STRINGREP (str_holder, str_length, record_numb);
    cyp$put_partial_line (print_file, str_holder
      (1, str_length), TRUE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$skip_lines (print_file, 2, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  PROCEND my_new_page_proc;


  PROCEDURE print_page_footer (print_file: cyt$file;
    VAR status: ost$status);

    VAR
      str_holder: string (3),
      str_length: integer,
      page_number: integer;

    cyp$put_partial_line (print_file, ' ', TRUE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    page_number := cyp$current_page_number (print_file);
    cyp$tab_file (print_file, 70, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_partial_line (print_file, 'PAGE ', FALSE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    STRINGREP (str_holder, str_length, page_number);
    cyp$put_partial_line (print_file, str_holder
      (1, str_length), TRUE, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  PROCEND print_page_footer;

  PUSH in_file_specs: [1 .. 4];
  in_file_specs^ [1].selector := cyc$file_kind;
  in_file_specs^ [1].file_kind := cyc$text_file;
```

```
    in_file_specs^ [2].selector := cyc$file_access;
    in_file_specs^ [2].file_access := cyc$read;
    in_file_specs^ [3].selector := cyc$file_existence;
    in_file_specs^ [3].file_existence := cyc$old_file;
    in_file_specs^ [4].selector := cyc$open_position;
    in_file_specs^ [4].open_position := cyc$beginning;

    PUSH out_file_specs: [1 .. 6];
    out_file_specs^ [1].selector := cyc$file_kind;
    out_file_specs^ [1].file_kind := cyc$display_file;
    out_file_specs^ [2].selector := cyc$file_access;
    out_file_specs^ [2].file_access := cyc$write;
    out_file_specs^ [3].selector := cyc$file_existence;
    out_file_specs^ [3].file_existence := cyc$new_or_old_file;
    out_file_specs^ [4].selector := cyc$page_width;
    out_file_specs^ [4].page_width := out_page_width;
    out_file_specs^ [5].selector := cyc$page_length;
    out_file_specs^ [5].page_length := out_page_length;
    out_file_specs^ [6].selector := cyc$new_page_procedure;
    out_file_specs^ [6].new_page_procedure.kind :=
          cyc$user_specified_procedure;
    out_file_specs^ [6].new_page_procedure.user_procedure :=
          ^my_new_page_proc;

    cyp$open_file (in_name, in_file_specs, in_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    WHILE TRUE DO
      cyp$get_partial_line (in_file, line, line_length, eol,
        status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
      CASE cyp$current_file_position (in_file) OF
      = cyc$end_of_information =

        cyp$position_display_page (out_file, footing_line_number,
          status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    print_page_footer (out_file, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    EXIT /main_loop/;

  = cyc$end_of_partition =
    file_numb := file_numb + 1;
    record_numb := 1;
    cyp$position_display_page (out_file, footing_line_number,
      status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    print_page_footer (out_file, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;

  = cyc$end_of_block =
    record_numb := record_numb + 1;
    cyp$position_display_page (out_file, footing_line_number,
      status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    print_page_footer (out_file, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;

  ELSE
    IF cyp$current_display_line (out_file) =
      footing_line_number THEN
        print_page_footer (out_file, status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
    IFEND;
    IF line_length > 0 THEN
      cyp$put_partial_line (out_file, line (1,
        line_length), eol, status);
```

```
          ELSE
            cyp$write_end_of_line (out_file, status);
          IFEND;
          IF NOT status.normal THEN
            EXIT /main_loop/;
          IFEND;
        CASEND;
      WHILEND /main_loop/;

      cyp$close_file (in_file, cyc$beginning, status);
      IF NOT status.normal THEN
        RETURN; {----->
      IFEND;
      cyp$close_file (out_file, cyc$asis, status);
      IF NOT status.normal THEN
        RETURN; {----->
      IFEND;

    PROCEND list;

  MODEND display_file_example;
```

## Page-Overflow Processing for Display Files

The procedures and functions described in this section are for display files only. For more information on the structure of display files, refer to Display File Structure earlier in this chapter.

When the number of lines written to a display file exceeds the file's specified page length, the line count is reset to zero, the page overflow mechanism is executed, and the line count begins again.

The page overflow mechanism is the sequence of events performed whenever the display-page length is exceeded. These events are the following:

- CYBIL I/O checks whether you have specified your own page-overflow procedure on the FILE_SPECIFICATIONS parameter of CYP$OPEN_FILE. If you have, your page-overflow procedure is called.
- In the absence of such a procedure, CYBIL I/O checks whether you specified the use of standard page headers on the FILE_SPECIFICATIONS parameter. If so, the header will be formatted and displayed.
- If neither a user-specified page-overflow procedure nor the standard header has been selected, a page eject is performed. (A format control character of 1 is automatically specified.)

The sequence of events may be approximated as follows:

```
get next display line
IF (line_count + 1) > display page length THEN
    line_count := 0
    IF user-specified new_page_procedure THEN
        call user-specified procedure
    ELSEIF standard procedure selected THEN
        perform display page eject
        format and display standard header
        skip 1 line
        ELSE

            perform display page eject

        IFEND

    IFEND

    display the display line
```

Standard page headers are either narrow format or wide format. If the page width established when the file was opened is greater than or equal to 132, the wide format is selected; otherwise, the narrow format is selected. (The page width is specified via the FILE_SPECIFICATIONS parameter on CYP$OPEN_FILE.)

The standard page headers are formatted as follows:

Narrow format:

| Line | Columns | Description of Header |
|---|---|---|
| 1 | 1-46 | String contained in the title field of the new_page_procedure record of the FILE_SPECIFICATIONS parameter when the file was opened. |
| | 48-55 | Date in mm/dd/yy format. |
| | 62-72 | 'PAGE ' and page number. |
| 2 | 1-22 | Operating system version. |
| | 48-59 | Time in system default format, or if no default is available, in hh:mm:ss format. |

Wide format (all on one line):

| Line | Columns | Description of Header |
|---|---|---|
| 1 | 1-46 | String contained in the title field of the new_page_procedure record of the FILE_SPECIFICATIONS parameter when the file was opened. |
| | 48-69 | Operating system version. |
| | 91-98 | Date in mm/dd/yy format. |
| | 110-121 | Time in system default format, or if no default is available, in hh:mm:ss format. |
| | 123-132 | 'PAGE ' and page number |

All fields in the standard headers are displayed left-justified with blank fill to the right.

Standard title lines can be produced from within user-specified new page procedures through the use of the CYP$DISPLAY_STANDARD_TITLE procedure, which is described in the next section.

The following procedures and functions may be used with display files only. These procedures and functions are described in greater detail on the following pages.

CYP$START_NEW_DISPLAY_PAGE

Calls the CYBIL I/O page overflow mechanism.

CYP$POSITION_DISPLAY_PAGE

Positions a display file at a specified line.

CYP$DISPLAY_STANDARD_TITLE

Formats and writes a standard title line to a display file.

CYP$DISPLAY_PAGE_EJECT

Positions a display file to the top of the next page.

CYP$DISPLAY_PAGE_LENGTH

Returns the page length associated with a display file.

CYP$CURRENT_DISPLAY_LINE

Returns the number of the current line within the current page of a display file.

CYP$CURRENT_PAGE_NUMBER

Returns the current page number of a display file.

## CYP$START_NEW_DISPLAY_PAGE

**Purpose** Calls the CYBIL I/O page overflow mechanism.

**Format** **CYP$START_NEW_DISPLAY_PAGE (display_file, status)**

**Parameters** **display_file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**
cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the last write to the display file was a partial line rather than a full line, that line is terminated and then a new display page is started.
- Attempting to use this procedure on a file not opened as a display file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OUTPUT_REQUEST in the status variable.

## CYP$POSITION_DISPLAY_PAGE

**Purpose** Positions a display file at a specified line.

**Format** **CYP$POSITION_DISPLAY_PAGE (display_file, line_number, status)**

**Parameters** **display_file**: cyt$file;

File identifier established when the file was opened.

**line_number**: cyt$page_length;

The line at which the file is to be positioned.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**
cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the value of the LINE_NUMBER parameter is greater than the current line number and less than or equal to page size, the file is positioned to that line on the current page. If LINE_NUMBER is less than or equal to the current line number, the page overflow mechanism is executed and the file is positioned at LINE_NUMBER on the next page. If LINE_NUMBER is greater than the page size, the page overflow mechanism is executed and the file will be positioned at the top of the next page.
- If the last write to the display file was a partial write, that line is terminated and then the display page is positioned.
- Attempting to use this procedure on a file not opened as a display file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OPERATION in the status variable.

## CYP$DISPLAY_STANDARD_TITLE

**Purpose** Formats and writes a standard title line to a display file.

**Format** **CYP$DISPLAY_STANDARD_TITLE (file, title, lines_after_title, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**title**: string (* <= cyc$title_size);

The text that is to appear in columns 1 thru 46 in the standard title.

**lines_after_title**: cyt$page_length;

The number of blank lines to appear between the standard title and the next display line.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- If the last write to the display file was a partial write, that display line is terminated and then the standard title is written.
- Attempting to use this procedure on a file not opened as a display file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OPERATION in the status variable.

## CYP$DISPLAY_PAGE_EJECT

**Purpose** Positions a display file to the top of the next page.

**Format** **CYP$DISPLAY_PAGE_EJECT (display_file, status)**

**Parameters** **display_file**: cyt$file;

File identifier established when the file was opened.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_output_request
cye$incorrect_operation

**Remarks**

- This procedure should only be called from a user-specified page overflow procedure.
- If the last write to the display file was a partial write, that line is terminated and then a display page eject is performed (with a format control character of 1 in column 1).
- Attempting to use this procedure on a file not opened as a display file will return CYE$INCORRECT_OPERATION in the status variable.
- Attempting to use this procedure on a file not opened for write or read write access will return CYE$INCORRECT_OPERATION in the status variable.

## CYP$DISPLAY_PAGE_LENGTH Function

| | |
|---|---|
| **Purpose** | Returns the page length associated with a display file. |
| **Format** | **CYP$DISPLAY_PAGE_LENGTH (display_file)**: cyt$page_length; |
| **Parameters** | **display_file**: cyt$file;<br>File identifier established when the file was opened. |
| **Remarks** | Attempting to use this function on a file not opened as a display file will return an undefined result. |

## CYP$CURRENT_DISPLAY_LINE Function

**Purpose** Returns the number of the current line within the current page of a display file.

**Format** **CYP$CURRENT_DISPLAY_LINE (display_file)**: cyt$page_length;

**Parameters** **display_file**: cyt$file;

File identifier established when the file was opened.

**Remarks**

- After any vertical spacing command (such as CYP$SKIP_LINES, CYP$DISPLAY_PAGE_EJECT, CYP$POSITION_DISPLAY_PAGE), the value returned is the next line to be displayed. After a write command (such as CYP$PUT_NEXT_LINE, CYP$PUT_PARTIAL_LINE, CYP$WRITE_END_OF_LINE), the value returned is the line just displayed.

- Attempting to use this function on a file not opened as a display file will return an undefined result.

## CYP$CURRENT_PAGE_NUMBER Function

| | |
|---|---|
| **Purpose** | Returns the current page number of a display file. |
| **Format** | **CYP$CURRENT_PAGE_NUMBER (display_file)**: integer; |
| **Parameters** | **display_file**: cyt$file; File identifier established when the file was opened. |
| **Remarks** | Attempting to use this function on a file not opened as a display file will return an undefined result. |

## Program Example Using Terminal I/O

The following example illustrates the use of display file procedures employing terminal I/O.

Example 1:

```
*copyc osd$default_pragmats

MODULE catenate_string_to_file;

*copyc cyp$get_next_line
*copyc cyp$put_partial_line
*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$current_file_position
*copyc ost$status


PROGRAM catenate_string_to_file
  (VAR status: ost$status);

  CONST
    input_file_name = 'INPUT',
    output_file_name = 'OUTPUT',
    prefix_string = 'INPUT TEXT = ''',
    suffix_string = '''';

  VAR
    chars_read: integer,
    ignore_status: ost$status,
    input_line: string (osc$max_string_size),
    input_file: cyt$file,
    input_file_specs: [static, read] array [1 .. 2] of
      cyt$file_specification := [[cyc$file_kind,
      cyc$text_file], [cyc$file_access, cyc$read]],
    output_file: cyt$file,
    output_file_specs: [static, read] array [1 .. 2] of
      cyt$file_specification := [[cyc$file_kind,
      cyc$text_file], [cyc$file_access, cyc$write]];



  status.normal := true;
```

```
cyp$open_file (input_file_name, ^input_file_specs,
  input_file, status);
IF status.normal THEN
  cyp$open_file (output_file_name, ^output_file_specs,
    output_file, status);
  IF status.normal THEN

    cyp$get_next_line (input_file, input_line, chars_read,
      status);
    IF status.normal THEN

    /read_write/
      WHILE (cyp$current_file_position (input_file) <>
            cyc$end_of_information) do
        cyp$put_partial_line (output_file, prefix_string,
          false, status);
        IF NOT status.normal THEN
          EXIT /read_write/;
        IFEND;
        cyp$put_partial_line (output_file, input_line (1,
          chars_read), false, status);
        IF NOT status.normal THEN
          EXIT /read_write/;
        IFEND;
        cyp$put_partial_line (output_file, suffix_string,
          true, status);
        IF NOT status.normal THEN
          EXIT /read_write/;
        IFEND;

        cyp$get_next_line (input_file, input_line,
          chars_read, status);
        IF NOT status.normal THEN
          EXIT /read_write/;
        IFEND;
      WHILEND /read_write/;
    IFEND;
  IFEND;
IFEND;
```

```
    cyp$close_file (input_file, cyc$default_open_position,
      ignore_status);
    cyp$close_file (output_file, cyc$default_open_position,
      ignore_status);

    PROCEND catenate_string_to_file;
  MODEND catenate_string_to_file;
```

# NOS/VE-Specific Procedures and Functions for CYBIL I/O 13

This chapter describes the procedures which are available only on the NOS/VE implementation of CYBIL I/O.

# NOS/VE-Specific Procedures and Functions for CYBIL I/O 13

Because they provide access to capabilities unique to NOS/VE, the procedures described in this chapter are intended to be used only with the NOS/VE implementation of CYBIL I/O. Programs meant to be portable between operating systems should minimize use of these procedures. [1]

The following are the NOS/VE-specific procedures, which are described on the following pages:

CYP$GET_FILE_IDENTIFIER

Returns a file's identifier that was established when the file was opened.

CYP$GET_BINARY_FILE_POINTER

Returns a pointer to the segment pointer used by CYBIL I/O.

CYP$OPEN_BINARY_FILE

Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening binary files.

CYP$OPEN_RECORD_FILE

Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening record files.

CYP$OPEN_TEXT_FILE

Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening text files.

CYP$OPEN_DISPLAY_FILE

Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening display files.

1. Currently, CYBIL I/O is only available for NOS/VE.

## CYP$GET_FILE_IDENTIFIER

**Purpose** Returns a file's identifier that can be used with the access method (AM) procedures contained in the NOS/VE program interface.

**Format** **CYP$GET_FILE_IDENTIFIER (file, file_identifier, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**file_identifier**: VAR of amt$file_identifier;

The file access identifier that uniquely identifies and is subsequently used to reference this instance of open.

**status**: ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open

**Remarks** The file access identifier returned by this procedure may be used on calls to access-method procedures such as AMP$FETCH which are specific to an instance of open.

## CYP$GET_BINARY_FILE_POINTER

**Purpose** Returns a pointer to the segment pointer used by CYBIL I/O.

**Format** **CYP$GET_BINARY_FILE_POINTER (file, binary_file_pointer, status)**

**Parameters** **file**: cyt$file;

File identifier established when the file was opened.

**binary_file_pointer**: VAR of ^amt$segment_pointer;

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$file_not_open
cye$incorrect_operation

**Remarks**

- This procedure allows you to direct the reading or writing of binary files when the CYP$PUT_NEXT_BINARY and CYP$GET_NEXT_BINARY procedures are not adequate. (For example, when pointer information is to be stored as part of the data to be written).
- Binary files are read and written using segment access. The CYP$GET_BINARY_FILE_POINTER procedure returns a pointer to the segment pointer that CYBIL I/O uses. CYBIL I/O gets the segment pointer as a sequence pointer (that is, the kind field of the segment pointer record is AMC$SEQUENCE_POINTER). The sequence pointer may be accessed by referencing the sequence_pointer field of the segment pointer. For example:

```
NEXT variable_pointer IN segment_pointer^.sequence_pointer;
```

## CYP$OPEN_BINARY_FILE

**Purpose** Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening binary files.

**Format** **CYP$OPEN_BINARY_FILE (file_name, file_access, file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, attribute_override, file_control, status)**

**Parameters** **file_name**: cyt$file_name;

Name of the file to be opened. On NOS/VE, a file name may be up to 512 characters in length, and can be a file reference.

**file_access**: cyt$file_access;

Permitted mode of access. The following values are defined:

CYC$READ

Read access only.

CYC$WRITE

Write access only.

CYC$READ_WRITE

Read or write access.

**file_attachment**: ^fst$attachment_options;

The attachment options in effect for the requested instance of open.

**default_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**mandated_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**attribute_validation**: ^fst$file_cycle_attributes;

A pointer to a record of required attribute values for the file or file cycle.

**attribute_override**: ^fst$file_cycle_attributes;

A pointer to a record of attribute values to be overridden for this instance of open.

**file_control**: VAR of cyt$file;

Returns a file identifier that must be used on all other calls to the file. This is a unique identifier used for the file while it is open; any other references to this file must include this identifier.

Attempting to call a CYBIL I/O procedure with an undefined or altered pointer will have unpredictable results.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**

cye$no_memory_to_open_file
cye$incorrect_open_request

**Remarks**

- The values specified for the FILE_ATTACHMENT, DEFAULT_CREATION_ATTRIBUTE, MANDATED_CREATION_ATTRIBUTE, ATTRIBUTE_VALIDATION, and ATTRIBUTE_OVERRIDE parameters are passed directly to the FSP$OPEN_FILE procedure.The values passed in these parameters are not validated or checked in any way.

- For more information on the FSP$OPEN_FILE procedure and its parameters, refer to the CYBIL for NOS/VE File Management manual.

## CYP$OPEN_RECORD_FILE

**Purpose** Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening record files.

**Format** **CYP$OPEN_RECORD_FILE (file_name, file_access, file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, attribute_override, file_control, status)**

**Parameters** **file_name**: cyt$file_name;

Name of the file to be opened. On NOS/VE, a file name may be up to 512 characters in length, and can be a file reference.

**file_access**: cyt$file_access;

Permitted mode of access. The following values are defined:

CYC$READ

Read access only.

CYC$WRITE

Write access only.

CYC$READ_WRITE

Read or write access.

**file_attachment**: ^fst$attachment_options;

The attachment options in effect for the requested instance of open.

**default_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**mandated_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**attribute_validation**: ^fst$file_cycle_attributes;

A pointer to a record of required attribute values for the file or file cycle.

**attribute_override**: ^fst$file_cycle_attributes;

A pointer to a record of attribute values to be overridden for this instance of open.

**file_control**: VAR of cyt$file;

Returns a file identifier that must be used on all other calls to the file. This is a unique identifier used for the file while it is open; any other references to this file must include this identifier.

Attempting to call a CYBIL I/O procedure with an undefined or altered pointer will have unpredictable results.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**

cye$no_memory_to_open_file
cye$incorrect_open_request

**Remarks**

- The values specified for the FILE_ATTACHMENT, DEFAULT_CREATION_ATTRIBUTE, MANDATED_CREATION_ATTRIBUTE, ATTRIBUTE_VALIDATION, and ATTRIBUTE_OVERRIDE parameters are passed directly to the FSP$OPEN_FILE procedure. The values passed in these parameters are not validated or checked in any way.

- For more information on the FSP$OPEN_FILE procedure and its parameters, refer to the CYBIL File Management manual.

## CYP$OPEN_TEXT_FILE

**Purpose** Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening text files.

**Format** **CYP$OPEN_TEXT_FILE (file_name, file_access, file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, attribute_override, file_control, status)**

**Parameters** **file_name**: cyt$file_name;

Name of the file to be opened. On NOS/VE, a file name may be up to 512 characters in length, and can be a file reference.

**file_access**: cyt$file_access;

Permitted mode of access. The following values are defined:

CYC$READ

Read access only.

CYC$WRITE

Write access only.

CYC$READ_WRITE

Read or write access.

**file_attachment**: ^fst$attachment_options;

The attachment options in effect for the requested instance of open.

**default_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**mandated_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**attribute_validation**: ^fst$file_cycle_attributes;

A pointer to a record of required attribute values for the file or file cycle.

**attribute_override**: ^fst$file_cycle_attributes;

A pointer to a record of attribute values to be overridden for this instance of open.

**file_control**: VAR of cyt$file;

Returns a file identifier that must be used on all other calls to the file. This is a unique identifier used for the file while it is open; any other references to this file must include this identifier.

Attempting to call a CYBIL I/O procedure with an undefined or altered pointer will have unpredictable results.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions**

cye$no_memory_to_open_file
cye$incorrect_open_request

**Remarks**

- The values specified for the FILE_ATTACHMENT, DEFAULT_CREATION_ATTRIBUTE, MANDATED_CREATION_ATTRIBUTE, ATTRIBUTE_VALIDATION, and ATTRIBUTE_OVERRIDE parameters are passed directly to the FSP$OPEN_FILE procedure.The values passed in these parameters are not validated or checked in any way.

- For more information on the FSP$OPEN_FILE procedure and its parameters, refer to the CYBIL for NOS/VE File Management manual.

## CYP$OPEN_DISPLAY_FILE

**Purpose** Utilizes the flexibility of the FSP$OPEN_FILE procedure when opening display files.

**Format** **CYP$OPEN_DISPLAY_FILE (file_name, file_access, file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, attribute_override, file_control, status)**

**Parameters** **file_name**: cyt$file_name;

Name of the file to be opened. On NOS/VE, a file name may be up to 512 characters in length, and can be a file reference.

**file_access**: cyt$file_access;

Permitted mode of access. The following values are defined:

CYC$READ

Read access only.

CYC$WRITE

Write access only.

CYC$READ_WRITE

Read or write access.

**file_attachment**: ^fst$attachment_options;

The attachment options in effect for the requested instance of open.

**default_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**mandated_creation_attribute**: ^fst$file_cycle_attributes;

A pointer to a record of file cycle attributes established for a file that is initially opened or created by this call.

**attribute_validation**: ^fst$file_cycle_attributes;

A pointer to a record of required attribute values for the file or file cycle.

**attribute_override**: ^fst$file_cycle_attributes;

A pointer to a record of attribute values to be overridden for this instance of open.

**file_control**: VAR of cyt$file;

Returns a file identifier that must be used on all other calls to the file. This is a unique identifier used for the file while it is open; any other references to this file must include this identifier.

Attempting to call a CYBIL I/O procedure with an undefined or altered pointer will have unpredictable results.

**status**: VAR of ost$status;

Status variable in which the completion status is returned.

**Conditions** cye$no_memory_to_open_file
cye$incorrect_open_request

**Remarks**

- The values specified for the FILE_ATTACHMENT, DEFAULT_CREATION_ATTRIBUTE, MANDATED_CREATION_ATTRIBUTE, ATTRIBUTE_VALIDATION, and ATTRIBUTE_OVERRIDE parameters are passed directly to the FSP$OPEN_FILE procedure. The values passed in these parameters are not validated or checked in any way.

- For more information on the FSP$OPEN_FILE procedure and its parameters, refer to the CYBIL for NOS/VE File Management manual.

# Appendixes

# Glossary A

## A

### Access Attribute

Characteristic of a variable that determines whether the variable can be both read and written. Specifying the access attribute READ makes the variable a read-only variable.

### Active Call Chain

List of calls that led to the current procedure.

### Active Segment Identifier (ASID)

A 16-bit field in the system virtual address (SVA) that uniquely identifies an active segment in the system. The segment number in the process virtual address (PVA), which is known locally to the program, is converted to the active segment identifier, which is known globally to the system. See also Process Virtual Address and System Virtual Address.

### Alphabetic Character

One of the following letters:

A through Z

a through z

See also Character and Alphanumeric Character.

### Alphanumeric Character

An alphabetic character or a digit. See also Character, Alphabetic Character, and Digit.

### ANSI

American National Standards Institute.

### ASCII

American Standard Code for Information Interchange.

### ASID

See Active Segment Identifier.

### Assignment Statement

A statement that assigns a value to a variable.

## B

### Batch Debugging

Debugging when the user has no direct control of debugging during program execution. Contrast with Interactive Debugging.

### BDP

Business data processing.

### Bit

A binary digit. A bit has a value of 0 or 1. See also Byte.

### Boolean

A kind of value that is evaluated as TRUE or FALSE.

### Break

The primary mechanism for Debug to gain control from an executing program. A break specifies an event and an address range such that when the event occurs within the address range, Debug takes control.

### Byte

A group of contiguous bits. For NOS/VE, one byte is equal to 8 bits. An ASCII character code uses the rightmost 7 bits of one byte.

### Byte Offset

A number corresponding to the number of bytes beyond the beginning of a line, procedure, module, or section.

## C

### Character

A letter, digit, space, or symbol that is represented by a code in one or more of the standard character sets.

It is also referred to as a byte when used as a unit of measure to specify block length, record length, and so forth.

A character can be a graphic character or a control character. A graphic character is printable; a control character is nonprintable and is used to control an input or output operation.

**Character Constant**

A fixed value that represents a single character.

**Comment**

Any character or sequence of characters that is preceded by an opening brace and terminated by a closing brace or an end of line. A comment is treated exactly as a space.

**Compilation Time**

The time at which a source program is translated by the compiler to an object program that can be loaded and executed. Contrast with Execution Time.

**Compiler**

A processor that accepts source code as input and generates object code as output.

**Condition Handler**

A procedure called when an exception condition occurs. Condition handler processing occurs after Debug processing if Debug mode is on. The procedure is called only if it has been established as the condition handler for the condition type and the condition occurs within its scope.

## D

**Delimiter**

The indicator that separates and organizes data.

**Digit**

One of the following characters:

0 1 2 3 4 5 6 7 8 9

## E

**Entry Point**

The point in a module at which execution of the module can begin.

**Event**

A condition, such as division by zero, that causes Debug to gain control.

### Execution Ring

The level of hardware privilege assigned to a procedure while it is executing.

### Execution Time

The time at which a compiled source program is executed. Also known as Run Time.

### Expression

Notation that represents a value. A constant or variable appearing alone, or combinations of constants, variables, and operators.

### External Reference

A call to an entry point in another module.

## F

### Field

A subdivision of a record that is referenced by name. For example, the field NORMAL in a record of type OST$STATUS called OLD_STATUS is referenced as follows:

```
OLD_STATUS.NORMAL
```

## I

### Integer Constant

One or more digits and, for hexadecimal integer constants, the following characters:

A B C D E F a b c d e f

A hexadecimal integer constant must begin with a digit. A preceding sign and subsequent radix are optional.

### Interactive Debugging

Debugging when the user has direct control of the debugging process. Contrast with Batch Debugging.

## L

### Load Module

A module reformatted for code sharing and efficient loading. When the user generates an object library, each object module in the module list is reformatted and written as a load module on the object library.

## M

### Machine Addressing

Use of actual machine addresses. Contrast with Module Addressing and Symbolic Addressing.

### Machine-Level Debugging

Debugging using machine-level terms such as machine addresses. A knowledge of machine architecture is required. Contrast with Symbolic Debugging.

### Module

A unit of text accepted as input by the loader, linker, or object library generator. See also Object Module and Load Module.

### Module Addressing

Use of addresses in terms of module and procedure names and an offset. Contrast with Machine Addressing and Symbolic Addressing.

## N

### Name

Combination of from 1 through 31 characters chosen from the following set:

- Alphabetic characters (A through Z and a through z).
- Digits (0 through 9).
- Special characters (#, @, $, and _).

The first character of a name cannot be a digit.

## O

### Object Code

Executable code produced by a compiler.

### Object Module

A compiler-generated unit containing object code and instructions for loading the object code. It is accepted as input by the system loader and the Object Library Generator.

## P

### Page

An allocatable unit of real memory.

### Pointer Variable

A CYBIL variable which contains the virtual address of a value.

### PP

Peripheral processor.

### Process Virtual Address (PVA)

The virtual address known locally by a program (or process). It is converted to a system virtual address (SVA) that is known globally by the system. It consists of a ring number, a segment number, and a byte number. The segment number is used to form the active segment identifier in the system virtual address. See also Active Segment Identifier and System Virtual Address.

### PVA

See Process Virtual Address.

## R

### Range

Value represented as two values separated by an ellipsis. The element is associated with the values from the first value through the second value. The first value must be less than or equal to the second value. For example:

```
1 .. 100
```

### Reserved Word

Word having a predefined meaning in a language. The user cannot define a new meaning or use for a reserved word.

### Ring

Level of hardware protection given a file or segment. A file is protected from unauthorized access by tasks executing in higher rings. See also Execution Ring.

### Run Time

See Execution Time.

## S

### SCL

System Command Language.

### Section

A storage area that contains variables with common access attributes (for example, read-only variables or read/write variables).

### Segment

One or more pages with the same attributes and hardware protection. A file.

### Source Code

Statements written for input to a compiler.

### SPT

System page table.

### Statement List

One or more statements separated by delimiters.

### String Constant

A constant that represents a string value. A sequence of characters delimited by apostrophes ('). An apostrophe can be included in the string by specifying two consecutive apostrophes.

### SVA

See System Virtual Address.

### Symbolic Addressing

Use of addresses in source program terms such as program names and line numbers. Contrast with Machine Addressing and Module Addressing.

### Symbolic Debugging

Debugging using source program terms such as line numbers and program names. Contrast with Machine-Level Debugging.

### System Virtual Address (SVA)

The virtual address known globally by the system. It is formed using the process virtual address, which is known locally by a program. It consists of an active segment identifier and a byte number. The system virtual address is translated into the real memory address. See also Active Segment Identifier and Process Virtual Address.

## T

### Traceback

A list of procedure names within a program, beginning with the currently executing procedure, proceeding backward through the sequence of called procedures, and ending with the main program.

## V

### Variable

Represents a data value.

### Variable Attribute

A characteristic of a variable. See also Access Attribute.

# Related Manuals B

All NOS/VE manuals and related hardware manuals are listed in table B-1. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/explain
```

## Ordering Printed Manuals

To order a printed Control Data manual, send an order form to:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

To obtain an order form or to get more information about ordering Control Data manuals, write to the above address or call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

## Accessing Online Manuals

To access the online version of a printed manual, log in to NOS/VE and enter the online title on the EXPLAIN command (table B-1 supplies the online titles). For example, to see the SCL Quick Reference online manual, enter:

```
/explain manual=SCL
```

The examples in some printed manuals exist also in the online Examples manual. To access this manual, enter:

```
/explain manual=examples
```

When EXAMPLES is listed in the Online Manuals column in table B-1, that manual is represented in the online Examples manual.

**Table B-1. Related Manuals**

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Site Manuals:** | | |
| CYBER Initialization Package (CIP) User's Handbook | 60457180 | |
| NOS/VE Accounting and Validation Utilities for Dual State Usage | 60458910 | |
| NOS/VE Accounting Analysis System Usage | 60463923 | |
| Family Administration for NOS/VE Usage | 60464513 | |
| NOS/VE Operations Usage | 60463914 | |
| NOS/VE System Analyst Reference Set System Performance and Maintenance Usage | 60463915 | |
| NOS/VE System Analyst Reference Set Network Interface Usage | 60463916 | |
| NOS/VE System Analyst Reference Set LCN Configuration and Network Management Usage | 60463917 | |
| CYBER 930 Computer System Basic Operations Usage | 60469560 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Site Manuals (Continued):** | | |
| Operator Messages for NOS/VE<br>Usage | 60464614 | |
| MAINTAIN_MAIL[2]<br>Usage | | MAIM |
| **SCL Manuals:** | | |
| Introduction to NOS/VE<br>Tutorial | 60464012 | |
| SCL for NOS/VE<br>Language Definition<br>Usage | 60464013 | EXAMPLES |
| SCL for NOS/VE<br>System Interface<br>Usage | 60464014 | EXAMPLES |
| NOS/VE File Editor<br>Tutorial/Usage | 60464015 | EXAMPLES |
| Terminal Definition for NOS/VE<br>Usage | 60464016 | |
| SCL for NOS/VE<br>Source Code Management<br>Usage | 60464313 | EXAMPLES |
| SCL for NOS/VE<br>Object Code Management<br>Usage | 60464413 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

2. To access this manual, you must be the administrator for MAIL/VE, have ring 4 privileges, and be at the system console.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **SCL Manuals (Continued):** | | |
| SCL for NOS/VE Quick Reference | 60464018 | SCL |
| SCL for NOS/VE Advanced File Management Tutorial | 60486412 | AFM_T |
| SCL for NOS/VE Advanced File Management Usage | 60486413 | AFM |
| SCL for NOS/VE Advanced File Management Summary | 60486419 | |
| EDIT_CATALOG Usage | | EDIT_ CATALOG |
| EDIT_CATALOG for NOS/VE Summary | 60487719 | |
| Screen Design Facility for NOS/VE Usage | 60488613 | SDF |
| Screen Formatting for NOS/VE Usage | 60488813 | EXAMPLES |
| Screen Formatting for NOS/VE Quick Reference | | SCREEN_ FORMATTING |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **CYBIL Manuals:** | | |
| CYBIL for NOS/VE Language Definition Usage | 60464113 | CYBIL and EXAMPLES |
| CYBIL for NOS/VE System Interface Usage | 60464115 | EXAMPLES |
| CYBIL for NOS/VE File Management Usage | 60464114 | EXAMPLES |
| CYBIL for NOS/VE Sequential and Byte-Addressable Files Usage | 60464116 | EXAMPLES |
| CYBIL for NOS/VE Keyed-File and Sort/Merge Interfaces Usage | 60464117 | EXAMPLES |
| **FORTRAN Manuals:** | | |
| FORTRAN for NOS/VE Tutorial | 60485912 | FORTRAN_T |
| FORTRAN for NOS/VE Language Definition Usage | 60485913 | EXAMPLES |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **FORTRAN Manuals (Continued):** | | |
| FORTRAN Version 2 for NOS/VE Language Definition Usage | 60487113 | EXAMPLES |
| FORTRAN for NOS/VE Topics for FORTRAN Programmers Usage | 60485916 | |
| FORTRAN for NOS/VE Quick Reference | | FORTRAN |
| FORTRAN Version 2 for NOS/VE Quick Reference | | VFORTRAN |
| FORTRAN for NOS/VE Summary | 60485919 | |
| **COBOL Manuals:** | | |
| COBOL for NOS/VE Tutorial | 60486012 | COBOL_T |
| COBOL for NOS/VE Usage | 60486013 | COBOL |
| COBOL for NOS/VE Summary | 60486019 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Other Compiler Manuals:** | | |
| APL for NOS/VE File Utilities Usage | 60485814 | |
| APL for NOS/VE Language Definition Usage | 60485813 | |
| BASIC for NOS/VE Usage | 60486313 | BASIC |
| BASIC for NOS/VE Summary Card | 60486319 | |
| LISP for NOS/VE Usage | 60486213 | |
| Pascal for NOS/VE Usage | 60485613 | PASCAL |
| Pascal for NOS/VE Summary Card | 60485619 | |
| Prolog for NOS/VE Usage | 60486713 | |
| Prolog for NOS/VE Quick Reference | 60486718 | PROLOG |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **VX/VE Manuals:** | | |
| VX/VE<br>An Introduction for UNIX Users<br>Tutorial/Usage | 60469980 | |
| VX/VE<br>Support Tools Guide<br>Tutorial | 60469800 | |
| VX/VE<br>Programmer Guide<br>Tutorial | 60469790 | |
| VX/VE<br>User Guide<br>Tutorial | 60469780 | |
| VX/VE<br>User Reference<br>Usage | 60469810 | |
| VX/VE<br>Programmer Reference<br>Usage | 60469820 | |
| VX/VE<br>Administrator Guide and Reference<br>Tutorial/Usage | 60469770 | |
| C/VE for NOS/VE<br>Quick Reference | | C |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **VX/VE Manuals (Continued):** | | |
| C/VE for NOS/VE<br>Usage | 60469830 | |
| DWB/VX<br>Introduction and User Reference<br>Tutorial/Usage | 60469890 | |
| DWB/VX<br>Text Formatters Guide<br>Usage | 60469900 | |
| DWB/VX<br>Macro Packages Guide<br>Usage | 60469910 | |
| DWB/VX<br>Preprocessors Guide<br>Usage | 60469920 | |
| **Data Management Manuals:** | | |
| IM/DM<br>Query, Report Writer, and<br>Command Procedures<br>Usage | 60489013 | |
| IM/DM<br>Commands for Query, Report Writer,<br>and Command Procedures<br>Usage | 60489018 | |
| IM/DM<br>Data Administration<br>Usage | 60489014 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

Table B-1. Related Manuals *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Data Management Manuals (Continued):** | | |
| IM/DM Application Programming Usage | 60489015 | |
| IM/Quick for NOS/VE Tutorial | 60485712 | |
| IM/Quick for NOS/VE Summary | 60485714 | |
| IM/Quick for NOS/VE Usage | | QUICK |
| IM/DM for NOS/VE Quick Reference | | IM_DM |
| **CDCNET Manuals:** | | |
| CDCNET Commands Quick Reference | 60000020 | |
| CDCNET Product Descriptions | 60460590 | |
| CDCNET Conceptual Overview | 60461540 | |
| CDCNET Configuration and Site Administration Guide | 60461550 | |
| CDCNET Network Operations | 60461520 | |
| CDCNET Network Performance Analyzer | 60461510 | |
| CDCNET Network Analysis | 60461590 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **CDCNET Manuals (Continued):** | | |
| CDCNET Diagnostic Messages | 60461600 | |
| CDCNET CYBIL Reference | 60462400 | |
| CDCNET System Programmer's Reference Volume 1 Base System Software | 60462410 | |
| CDCNET System Programmer's Reference Volume 2 Network Management Entities and Layer Interfaces | 60462420 | |
| CDCNET System Programmer's Reference Volume 3 Network Protocols | 60462430 | |
| CDCNET Access | 60463830 | |
| CDCNET Terminal Interface Usage | 60463850 | |
| CDCNET Terminal Interface for NOS/VE Quick Reference | | CDCNET_ ACCESS |
| CDCNET Batch Device User Guide | 60463863 | CDCNET_ BATCH |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Migration Manuals:** | | |
| Migration from IBM to NOS/VE Tutorial/Usage | 60489507 | MIGRATE_ IBM |
| Migration from NOS to NOS/VE Tutorial/Usage | 60489503 | MIGRATE_ NOS |
| Migration from NOS to NOS/VE Standalone Tutorial/Usage | 60489504 | |
| Migration from NOS/BE to NOS/VE Tutorial/Usage | 60489505 | MIGRATE_ NOSBE |
| Migration from NOS/BE to NOS/VE Standalone Tutorial/Usage | 60489506 | |
| Migration from VAX/VMS to NOS/VE Tutorial/Usage | 60489508 | MIGRATE_ VAX |
| **Miscellaneous Manuals:** | | |
| Control Data CONNECT User's Guide | 60462560 | |
| CYBER Online Text for NOS/VE Usage | 60488403 | CONTEXT |
| CONTEXT Summary Card | 60488419 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Miscellaneous Manuals (Continued):** | | |
| Debug for NOS/VE Usage | 60488213 | |
| Debug for NOS/VE Quick Reference | | DEBUG |
| Diagnostic Messages for NOS/VE Usage | 60464613 | MESSAGES |
| MAIL/VE Summary Card | 60464519 | |
| MAIL/VE Usage | | MAIL_VE |
| NOS/VE Examples Usage | | EXAMPLES |
| NOS/VE System Information | | NOS_VE |
| Programming Environment for NOS/VE Usage | | ENVIRON-MENT |
| Programming Environment for NOS/VE Summary | 60486819 | |
| Remote Host Facility Usage | 60460620 | |
| Applications Directory | 60455370 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

*(Continued)*

**Table B-1. Related Manuals** *(Continued)*

| Manual Title | Publication Number | Online Manuals[1] |
|---|---|---|
| **Hardware Manuals:** | | |
| CYBER Installation Package (CIP) Handbook | 60457180 | |
| CYBER 170 Computer Systems Models 825, 835, and 855 General Description Hardware Reference | 60459960 | |
| CYBER 170 Computer Systems, Models 815, 825, 835, 845, and 855 CYBER 180 Models 810, 830, 835, 840, 845, 850, 855, and 860 Codes Booklet | 60458100 | |
| CYBER 170 Computer Systems, Models 815, 825, 835, 845, and 855 CYBER 180 Models 810, 830, 835, 840, 845, 850, 855, and 860 Maintenance Register Codes Booklet | 60458110 | |
| Virtual State Volume II Hardware Reference | 60458890 | |
| 7221-1 Intelligent Small Magnetic Tape Subsystem Reference | 60461090 | |
| 7021-31/32 Advanced Tape Subsystem Reference | 60449600 | |
| HPA/VE User's Reference Subsystem Reference | 60461930 | |

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

# Character Set C

This appendix lists the ASCII character set (refer to table C-1).

NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.4-1977). NOS/VE represents each 7-bit ASCII code in an 8-bit byte. These 7 bits are right justified in each byte. For ASCII characters, the eighth or leftmost bit is always zero. However, in NOS/VE the leftmost bit can also be used to define an additional 128 characters.

If you want to define additional non-ASCII characters, be certain that the leftmost bit is available in your current working environment. The full screen applications (such as the EDIT_FILE utility, the EDIT_CATALOG utility, and the programming language environments) already use this bit for special purposes. Therefore, these applications accept only the standard ASCII characters. In applications in which the leftmost bit is not used, however, you are free to use it to define the interpretation of each character as you wish.

**Table C-1. ASCII Character Set**

| Decimal Code | Hexa-decimal Code | Octal Code | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|---|
| 000 | 00 | 000 | NUL | Null |
| 001 | 01 | 001 | SOH | Start of heading |
| 002 | 02 | 002 | STX | Start of text |
| 003 | 03 | 003 | ETX | End of text |
| 004 | 04 | 004 | EOT | End of transmission |
| 005 | 05 | 005 | ENQ | Enquiry |
| 006 | 06 | 006 | ACK | Acknowledge |
| 007 | 07 | 007 | BEL | Bell |
| 008 | 08 | 010 | BS | Backspace |
| 009 | 09 | 011 | HT | Horizontal tabulation |
| 010 | 0A | 012 | LF | Line feed |
| 011 | 0B | 013 | VT | Vertical tabulation |
| 012 | 0C | 014 | FF | Form feed |
| 013 | 0D | 015 | CR | Carriage return |
| 014 | 0E | 016 | SO | Shift out |
| 015 | 0F | 017 | SI | Shift in |
| 016 | 10 | 020 | DLE | Data link escape |
| 017 | 11 | 021 | DC1 | Device control 1 |
| 018 | 12 | 022 | DC2 | Device control 2 |
| 019 | 13 | 023 | DC3 | Device control 3 |
| 020 | 14 | 024 | DC4 | Device control 4 |
| 021 | 15 | 025 | NAK | Negative acknowledge |
| 022 | 16 | 026 | SYN | Synchronous idle |
| 023 | 17 | 027 | ETB | End of transmission block |
| 024 | 18 | 030 | CAN | Cancel |
| 025 | 19 | 031 | EM | End of medium |
| 026 | 1A | 032 | SUB | Substitute |
| 027 | 1B | 033 | ESC | Escape |
| 028 | 1C | 034 | FS | File separator |
| 029 | 1D | 035 | GS | Group separator |
| 030 | 1E | 036 | RS | Record separator |
| 031 | 1F | 037 | US | Unit separator |
| 032 | 20 | 040 | SP | Space |
| 033 | 21 | 041 | ! | Exclamation point |
| 034 | 22 | 042 | " | Quotation marks |
| 035 | 23 | 043 | # | Number sign |
| 036 | 24 | 044 | $ | Dollar sign |
| 037 | 25 | 045 | % | Percent sign |
| 038 | 26 | 046 | & | Ampersand |
| 039 | 27 | 047 | ’ | Apostrophe |

(Continued)

**Table C-1. ASCII Character Set** (Continued)

| Decimal Code | Hexa-decimal Code | Octal Code | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|---|
| 040 | 28 | 050 | ( | Opening parenthesis |
| 041 | 29 | 051 | ) | Closing parenthesis |
| 042 | 2A | 052 | * | Asterisk |
| 043 | 2B | 053 | + | Plus |
| 044 | 2C | 054 | , | Comma |
| 045 | 2D | 055 | − | Hyphen |
| 046 | 2E | 056 | . | Period |
| 047 | 2F | 057 | / | Slant |
| 048 | 30 | 060 | 0 | Zero |
| 049 | 31 | 061 | 1 | One |
| 050 | 32 | 062 | 2 | Two |
| 051 | 33 | 063 | 3 | Three |
| 052 | 34 | 064 | 4 | Four |
| 053 | 35 | 065 | 5 | Five |
| 054 | 36 | 066 | 6 | Six |
| 055 | 37 | 067 | 7 | Seven |
| 056 | 38 | 070 | 8 | Eight |
| 057 | 39 | 071 | 9 | Nine |
| 058 | 3A | 072 | : | Colon |
| 059 | 3B | 073 | ; | Semicolon |
| 060 | 3C | 074 | < | Less than |
| 061 | 3D | 075 | = | Equals |
| 062 | 3E | 076 | > | Greater than |
| 063 | 3F | 077 | ? | Question mark |
| 064 | 40 | 100 | @ | Commercial at |
| 065 | 41 | 101 | A | Uppercase A |
| 066 | 42 | 102 | B | Uppercase B |
| 067 | 43 | 103 | C | Uppercase C |
| 068 | 44 | 104 | D | Uppercase D |
| 069 | 45 | 105 | E | Uppercase E |
| 070 | 46 | 106 | F | Uppercase F |
| 071 | 47 | 107 | G | Uppercase G |
| 072 | 48 | 110 | H | Uppercase H |
| 073 | 49 | 111 | I | Uppercase I |
| 074 | 4A | 112 | J | Uppercase J |
| 075 | 4B | 113 | K | Uppercase K |
| 076 | 4C | 114 | L | Uppercase L |
| 077 | 4D | 115 | M | Uppercase M |
| 078 | 4E | 116 | N | Uppercase N |
| 079 | 4F | 117 | O | Uppercase O |

(Continued)

Table C-1. ASCII Character Set (Continued)

| Decimal Code | Hexa-decimal Code | Octal Code | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|---|
| 080 | 50 | 120 | P | Uppercase P |
| 081 | 51 | 121 | Q | Uppercase Q |
| 082 | 52 | 122 | R | Uppercase R |
| 083 | 53 | 123 | S | Uppercase S |
| 084 | 54 | 124 | T | Uppercase T |
| 085 | 55 | 125 | U | Uppercase U |
| 086 | 56 | 126 | V | Uppercase V |
| 087 | 57 | 127 | W | Uppercase W |
| 088 | 58 | 130 | X | Uppercase X |
| 089 | 59 | 131 | Y | Uppercase Y |
| 090 | 5A | 132 | Z | Uppercase Z |
| 091 | 5B | 133 | [ | Opening bracket |
| 092 | 5C | 134 | \ | Reverse slant |
| 093 | 5D | 135 | ] | Closing bracket |
| 094 | 5E | 136 | ^ | Circumflex |
| 095 | 5F | 137 | _ | Underline |
| 096 | 60 | 140 | ` | Grave accent |
| 097 | 61 | 141 | a | Lowercase a |
| 098 | 62 | 142 | b | Lowercase b |
| 099 | 63 | 143 | c | Lowercase c |
| 100 | 64 | 144 | d | Lowercase d |
| 101 | 65 | 145 | e | Lowercase e |
| 102 | 66 | 146 | f | Lowercase f |
| 103 | 67 | 147 | g | Lowercase g |
| 104 | 68 | 150 | h | Lowercase h |
| 105 | 69 | 151 | i | Lowercase i |
| 106 | 6A | 152 | j | Lowercase j |
| 107 | 6B | 153 | k | Lowercase k |
| 108 | 6C | 154 | l | Lowercase l |
| 109 | 6D | 155 | m | Lowercase m |
| 110 | 6E | 156 | n | Lowercase n |
| 111 | 6F | 157 | o | Lowercase o |
| 112 | 70 | 160 | p | Lowercase p |
| 113 | 71 | 161 | q | Lowercase q |
| 114 | 72 | 162 | r | Lowercase r |
| 115 | 73 | 163 | s | Lowercase s |
| 116 | 74 | 164 | t | Lowercase t |
| 117 | 75 | 165 | u | Lowercase u |
| 118 | 76 | 166 | v | Lowercase v |
| 119 | 77 | 167 | w | Lowercase w |

(Continued)

**Table C-1. ASCII Character Set (Continued)**

| Decimal Code | Hexa-decimal Code | Octal Code | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|---|
| 120 | 78 | 170 | x | Lowercase x |
| 121 | 79 | 171 | y | Lowercase y |
| 122 | 7A | 172 | z | Lowercase z |
| 123 | 7B | 173 | { | Opening brace |
| 124 | 7C | 174 | \| | Vertical line |
| 125 | 7D | 175 | } | Closing brace |
| 126 | 7E | 176 | ~ | Tilde |
| 127 | 7F | 177 | DEL | Delete |

# Reserved Words D

The reserved words in CYBIL are listed next.

ALIAS
ALIGNED
ALLOCATE
AND
ARRAY
BEGIN
BOOLEAN
BOUND
CASE
CASEND
CAT
CELL
CHAR
CHKALL
CHKNIL
CHKRNG
CHKSUB
CHKTAG
CHR
COMMENT
COMPILE
CONST
CYCLE
DIV
DO
DOWNTO
EJECT
ELSE
ELSEIF
END
EXIT
FALSE
FMT
FOR
FOREND
FREE
FUNCEND
FUNCTION
HEAP
IF
IFEND
IN
INLINE
INTEGER
LEFT
LIBRARY
LIST
LISTALL
LISTCTS
LISTEXT
LISTOBJ
LOWERBOUND
LOWERVALUE
MOD
MODEND
MODULE
NEWTITLE
NEXT
NIL
NOCOMPILE
NOT
OF
OFF
OLDTITLE
ON
OR
ORD
PACKED
POP
PRED
PROCEDURE
PROCEND
PROGRAM
PUSH
READ
REAL
RECEND
RECORD
REL
REP
REPEAT
RESET
RETURN
RIGHT
SECTION
SEQ
SET
SKIP
SPACING
STATIC
STRING
STRINGREP
STRLENGTH
SUCC
THEN
TITLE
TO
TRUE
TYPE
UNTIL
UPPERBOUND
UPPERVALUE
VAR
WHILE
WHILEND
WRITE
XDCL
XOR
XREF
#ADDRESS
#CALLER_ID
#COMPARE_SWAP
#CONVERT_POINTER_TO_PROCEDURE

#FREE_
RUNNING_
CLOCK
#GATE
#HASH_SVA
#INLINE
#KEYPOINT
#LOC
#OFFSET
#PREVIOUS_
SAVE_AREA

#PTR
#PURGE_
BUFFER
#READ_
REGISTER
#REL
#RING
#SCAN
#SEGMENT
#SEQ
#SIZE

#SPOIL
#TRANSLATE
#UNCHECKED_
CONVERSION
#WRITE_REGISTER
$CHAR
$INTEGER
$REAL

# Data Representation in Memory E

This appendix describes how the CYBIL data types are stored in memory. Table E-1 summarizes, for quick reference, the simple, common data types. Following the table, detailed descriptions are given for each type. Finally, examples illustrate how a record is stored with variations in packing, positioning, and alignment. Although table E-1 gives sufficient detail for most cases, you should also read the description for the particular type you are using, especially if it is a more complex pointer or structured type.

The smallest unit of memory in NOS/VE is a bit. There are 8 bits to a byte and 8 bytes to a 64-bit word. Both bytes and words can be addressed directly (an 8-bit byte is synonymous with a cell).

Table E-1 summarizes how different data types are represented in memory. The last two columns indicate how the specified data type is stored if it is a component of an unpacked or packed structure. The word byte means the data type would be stored in the first available byte; bit means it would be stored in the first available bit.

**Table E-1. Data Representation in Memory**

| Type | Size | Unpacked Alignment | Packed Alignment |
|---|---|---|---|
| Integer | 8 bytes | Byte | Byte |
| Character | 1 byte | Byte | Bit |
| Boolean | 1 bit | Right-justified in a byte | Bit |
| Ordinal | As needed for components | Right-justified in a byte | Bit if ≤ 57 components; byte if > 57 components |
| Subrange | As needed for components | Right-justified in a byte | Bit if ≤ 57 components; byte if > 57 components |
| Real | 8 bytes | Byte | Byte |
| Cell | Byte | Byte | Byte |
| Fixed pointer | 6 bytes | Byte | Byte |
| Fixed relative pointer | 4 bytes | Byte | Byte |
| String | 1 byte for each character | Byte | Byte |
| Array/ Record | Depends on type of components | Byte | Components are unaligned |
| Set | As needed for components | Right-justified in a byte | Bit if ≤ 57 components; byte if > 57 components |

## CYBIL Data Types

In the descriptions that follow, the alignment of the type may differ depending on whether it is the type of a declared variable (which stands by itself) or the type of a component within a structure (such as a field within a record variable). For example, a variable that is declared as an integer is aligned on a word boundary. But a field that is declared to be of integer type and is within a record is aligned on a byte. If the description applies to a variable of the given type, it states that specifically. Otherwise, the description applies only to a component of that type which is itself within another structure (or can be within another structure).

The phrase word-aligned means the type is aligned on a word boundary; likewise, byte-aligned means the type is aligned on a byte boundary.

### Integers

An integer variable is allocated 64 bits and is aligned on a word boundary. An integer-type component within a structure is aligned on a byte boundary; this is true regardless of whether the structure is packed or unpacked.

### Characters

A character variable is allocated 8 bits and is aligned on the rightmost byte of a word. Within a structure, an unpacked character type is byte-aligned; a packed character type is bit-aligned.

### Boolean

A boolean variable is allocated 1 byte; it is byte-aligned and right-justified in a word. Within a structure, an unpacked boolean type is allocated 1 byte and is byte-aligned. A packed boolean type is allocated 1 bit and is bit-aligned.

The internal value used for FALSE is 0; the value used for TRUE is 1.

## Ordinals

An ordinal type is treated as a subrange type from 0 through the total number of elements minus 1 (that is, 0..n-1 where n represents the total number of elements in the ordinal). For further information, refer to the description of subranges.

## Subranges

A subrange variable is allocated 8 bytes if its lower bound is negative. If its lower bound is 0 or positive, it is allocated from 1 to 8 bytes, depending on the value of the upper bound. It is byte-aligned on the rightmost byte of a word.

Within a structure, an unpacked subrange type is allocated the same as a subrange variable; however, it need not be aligned on the rightmost byte of a word although it is byte-aligned.

A packed subrange type is allocated enough memory to hold the subrange in bits. The exact length can be calculated using the following formula. This formula assumes the range is given as A..B (that is, A is the lower bound and B is the upper bound).

If A ≥ 0, LENGTH = CEILING (LOG2 (B+1) )

If A < 0, LENGTH = 1 + CEILING (LOG2 (MAX (ABS(A), B+1) ) )

A packed subrange type is bit-aligned if it contains 57 or less components; it is byte-aligned if it contains more than 57 components.

The maximum integer subrange is −7FFFFFFFFFFFFFFF hexadecimal through 7FFFFFFFFFFFFFFF hexadecimal.

## Reals

A real variable is allocated 64 bits and is word-aligned. Within a structure, a real type is byte-aligned; this is true regardless of whether the structure is packed or unpacked.

## Cells

A cell type is allocated 1 byte and is byte-aligned.

## Pointers

A pointer is composed of the address of the first byte of the object to which it points, plus any information that is needed to describe the object. The address field of the pointer is a 6-byte process virtual address (PVA) that is always byte-aligned. It has the following format:

```
PROCESS_VIRTUAL_ADDRESS = PACKED RECORD
  RING_NUMBER: 0 .. 15, (4 bits, unsigned)
SEGMENT_NUMBER: 0 .. 4095, (12 bits, unsigned)
BYTE_NUMBER: HALF_INTEGER, (32 bits, signed)
RECEND;
```

HALF_INTEGER is defined as the subrange from −80000000 hexadecimal through 7FFFFFFF hexadecimal; that is,

```
HALF_INTEGER = -80000000(16) .. 7FFFFFFF(16);
```

Generally, a pointer to an object with a fixed size contains only the PVA; the single exception is a pointer to a sequence which is described later in this section. A pointer to an object with an adaptable size contains the PVA and a descriptor for the adaptable object (described later). The descriptor immediately follows the PVA.

A pointer variable that occupies 8 or more bytes is word-aligned on the left. A pointer variable of less than 8 bytes is right-justified in a word. Within a structure, a pointer type is byte-aligned; this is true regardless of whether the structure is packed or unpacked.

### NIL Pointer Constant

The NIL pointer constant is defined as follows:

```
NIL: PROCESS_VIRTUAL_ADDRESS := [0F(16), 0FFF(16), 80000000(16)]
```

In other words, using the record format shown earlier for a process virtual address, the ring number is set to 0F hexadecimal, the segment number to 0FFF hexadecimal, and the byte number to 80000000 hexadecimal.

## Adaptable Pointers

The descriptors for adaptable objects of pointers are byte-aligned and have the following formats for adaptable strings, arrays, sequences, user heaps, and records.

- An adaptable string descriptor is a 2-byte field with the following format:

  ```
  ADAPTABLE_STRING_SIZE: 0 .. 65535
  ```

  This field indicates the length of the string in bytes. The length can be in the range of 0 through 65,535.

- An adaptable array descriptor is a 12-byte record with the following format:

  ```
  ARRAY_DESCRIPTOR = RECORD
    ARRAY_SIZE: HALF_INTEGER,  (in bits or bytes)
  LOWER_BOUND: HALF_INTEGER,
    ELEMENT_SIZE: HALF_INTEGER, (in bits or bytes)
  RECEND;
  ```

  When the array is unpacked, the ARRAY_SIZE and ELEMENT_SIZE fields are both in bytes. When the array is packed, they are both in bits.

- An adaptable sequence descriptor with its pointer is a 14-byte record with the following format:

  ```
  SEQUENCE_POINTER = RECORD
    POINTER_SEQUENCE: PROCESS_VIRTUAL_ADDRESS,
    LIMIT: HALF_INTEGER,
    AVAILABLE: HALF_INTEGER,
  RECEND;
  ```

  The LIMIT field is an offset to the top of the sequence. The AVAILABLE field is an offset to the next available location in the sequence.

  This format is the same as for a pointer to a fixed sequence.

- An adaptable user heap descriptor is a 4-byte field with the following format:

  ```
  ADAPTABLE_USER_HEAP_SIZE: HALF_INTEGER;
  ```

  This field indicates the maximum length of the structure in bytes.

- An adaptable record descriptor is the descriptor of the adaptable field within the record (as previously described for adaptable types).

## Pointers to Sequences

A pointer to a sequence is a 14-byte record with the following format:

```
SEQUENCE_POINTER = RECORD
  POINTER_SEQUENCE: PROCESS_VIRTUAL_ADDRESS,
  LIMIT: HALF_INTEGER,
  AVAILABLE: HALF_INTEGER,
RECEND;
```

The LIMIT field is an offset to the top of the sequence. The AVAILABLE field is an offset to the next available location in the sequence.

This format is the same as for a pointer to an adaptable sequence.

## Pointers to Bound Variant Records

A pointer to a bound variant record consists of a 6-byte PVA, followed by a 4-byte size descriptor with the following format:

```
BOUND_VARIANT_RECORD_SIZE: HALF_INTEGER;
```

## Pointers to Procedures

A pointer to a procedure is a 12-byte record with the following format:

```
PROC_POINTER = RECORD
  POINTER_TO_PROCEDURE_DESCRIPTOR: PROCESS_VIRTUAL_ADDRESS,
  STATIC_LINK_OR_NIL: PROCESS_VIRTUAL_ADDRESS,
RECEND;
```

The first field in the procedure pointer is a pointer to the procedure descriptor in the binding section. This descriptor consists of two fields: a code base pointer and a binding section pointer. For further information about these pointers and the binding section, refer to volume II of the virtual state hardware reference manual.

The second field in the procedure pointer is the static link. A procedure declared at the outermost level of a module does not require a static link and, therefore, the NIL procedure pointer is used instead. This ensures that pointer comparison always works. The NIL procedure pointer constant is defined as follows:

```
NIL_PROC_POINTER: PROC_POINTER := [POINTER_TO_NIL_PROCEDURE_
  DESCRIPTOR,NIL];
```

where the NIL procedure descriptor points to an execution-time library procedure that handles a call through a NIL procedure pointer as an error.

## Relative Pointers

A relative pointer is a 4-byte field with the following format:

```
RELATIVE_ADDRESS = 0 .. 0FFFFFFFF(16);
```

This field gives the byte offset of the object field from the start of the parent variable. This relative address can be in the range of 0 through 0FFFFFFFF hexadecimal.

A relative pointer is byte-aligned.

### Adaptable Relative Pointers

A relative pointer to an adaptable type object is the 4-byte relative address plus a descriptor for the adaptable object. This descriptor immediately follows the relative address field. Descriptors for adaptable relative pointer types have the same alignment and formats as described previously under Adaptable Pointers.

### Relative Pointers to Sequences

A relative pointer to a sequence is a 12-byte record with the following format:

```
RELATIVE_POINTER_TO_SEQUENCE = RECORD
  RELATIVE_POINTER: RELATIVE_ADDRESS,
  LIMIT: HALF_INTEGER,
  AVAILABLE: HALF_INTEGER,
RECEND;
```

This format is the same for both fixed and adaptable sequences.

### Relative Pointers to Bound Variant Records

A relative pointer to a bound variant record is the 4-byte relative address followed by a 4-byte size descriptor with the following format:

```
BOUND_VARIANT_RECORD_SIZE: HALF_INTEGER;
```

## Strings

A string variable is allocated the same number of bytes as there are characters in the string. If the variable occupies 8 or more bytes, it is word-aligned on the left. If it is less than 8 bytes, it is word-aligned on the right.

Within a structure, a string type is byte-aligned; this is true regardless of whether the structure is packed or unpacked.

The maximum number of characters allowed in a string is 65,535.

## Arrays

An array variable is treated as a contiguous list of elements of the array's component type with space allocated for each element according to that component type. The array variable is word-aligned on the left.

Within a structure, an unpacked array type is allocated the same as an array variable and its elements are always aligned.

A packed array type is allocated similarly but its elements are not aligned. The array itself is aligned on a byte boundary if the type of its component elements would normally start on a byte boundary, or if the array is larger than 57 bits. If the component type of the array is byte-aligned, it occupies an integral number of bytes.

In general, the size of an array is limited only by the availability of memory. However, the maximum size of an array is the size of a segment (that is, 7FFFFFFF hexadecimal).

The size of an array of aligned records is a multiple of the base specified in the alignment parameter (ALIGNED offset MOD base) in the record declaration.

## Records

A record variable is treated as a contiguous list of the record fields and is allocated space according to the types of those individual fields. If it occupies more than one word, it is word-aligned on the left in the first word; a record that occupies less than one word is word-aligned on the right.

An unpacked record type is allocated space the same as a record variable and its fields are aligned. The record itself is aligned on the boundary of the maximum alignment of any of its fields. For example, if one field has an alignment of 0 MOD 8[1] and another field has an alignment of 0 MOD 16, the alignment with the larger base (0 MOD 16) is applied to the entire record; it is aligned on an even word boundary.

A packed record type is allocated similarly but its fields are not aligned. Like an unpacked record type, the record itself is aligned on the maximum alignment of its fields. However, if the record is more than 57 bits, it must be byte-aligned at least.

The length of a packed record depends on the length and alignment of its fields. The way a packed record is used (for example, as a record variable by itself, as a field in a larger record, or as an element of an array) does not affect how it is represented in memory. Thus, all occurrences of a packed record have the same length and alignment regardless of how it is used.

1. Record alignment parameters are described under Records in chapter 4.

If a field within a record contains the alignment parameter (ALIGNED offset MOD base), CYBIL first attempts to satisfy the offset value within the word being allocated. If the first field of a record is aligned, the entire record takes on the base specified in the alignment parameter. However, if other fields are also aligned, the record takes on the maximum base from all the fields in the record.

The record fields are allocated consecutively, subject to the alignment restrictions on them. If a record is byte-aligned, it occupies an integral number of bytes.

If a field in a packed or unpacked record is a character, boolean, ordinal, subrange, or set type and that field is not the subject of a pointer or a reference parameter, that field can be expanded. This means that if the field is followed by unused bits that extend to the next used field of the record (or to the end of the record), the expandable field can be enlarged to include as many bits as possible up to the next field. Character, boolean, ordinal, and subrange type fields can expand up to 32 bits. A set type that contains less than 57 elements can expand up to 57 bits, but only if it can expand to the next used field of the record. A set that contains more than 57 elements can expand to the next byte boundary or to the next field, whichever comes first. The content of the unused bits is undefined.

## Sets

A set is represented by enough contiguous bits to hold the total number of elements in the set's type. The leftmost bit corresponds to the first element of the set's type, the next bit corresponds to the second element, and so on.

A set variable is allocated a field of enough bytes to contain all the set's elements. If the field fits in a word, it is word-aligned on the right; otherwise, it is word-aligned on the left.

Within a structure, an unpacked set type is allocated the same as a set variable. The field allocated is byte-aligned.

A packed set type that contains more than 57 elements is treated as an unpacked set type. A packed set type that contains 57 or less elements is allocated a field with the number of bits (rather than bytes) necessary to hold the elements of the set. This field is bit-aligned.

When the field is allocated in bytes rather than bits, the field may be larger than is necessary to hold the total number of elements in the base type of the set. In this case, the elements are right-justified in the field and unused bits to the left of the elements are set to 0.

The maximum number of elements allowed in a set is 32,767.

## Sequences

A sequence is allocated enough storage space to hold the spans (the number of occurrences of each type) you specified when you declared the sequence. You can determine the total amount of storage required by using the #SIZE function on each span and adding the results.

A sequence has the following format:

```
SEQUENCE = RECORD
  DATA_AREA: SPACE,
RECEND;
```

## Heaps

A user-declared heap is allocated enough storage space to hold the spans (the number of occurrences of each type) you specified when you declared the heap, plus certain controlling information for each span. You can determine the total amount of storage required by using the #SIZE function on each span and adding the results. Then, add a 16-byte header for each repetition count (as specified by the REP number OF parameter) on each span.

The user heap and the default heap both have the following format:

```
HEAP = PACKED RECORD
  BLOCK_STATUS:       (AVAIL, USED),
  SIZE:               0..7FFFFFFF(16),
  FORWARD_FREE_LINK:  0..0FFFFFFFF(16),
  BACKWARD_LINK:      0..0FFFFFFFF(16),
  FORWARD_LINK:       0..0FFFFFFFF(16),
    DATA_AREA:        SPACE,
RECEND;
```

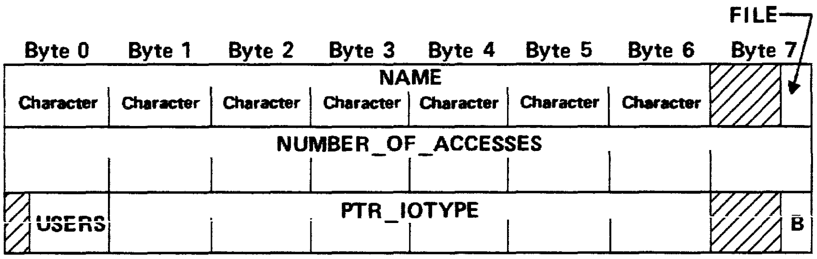
## Examples

The following examples show how a record would look in memory in various formats: first unpacked, then packed, packed with some positioning changes, and finally aligned. The memory shown here is in 8-byte words, but because bytes can be addressed individually, it's possible the record could start at any byte (if it is not aligned otherwise).

The unpacked record is:

```
TYPE
  table = record
    name: string(7),
    file: (bi, di, lg, pr),
    number_of_accesses: integer,
    users: 0 .. 100,
    ptr_iotype: ^iotype,
    b: boolean,
  recend;
```

This record would appear in memory as follows (slashes indicate unused memory):

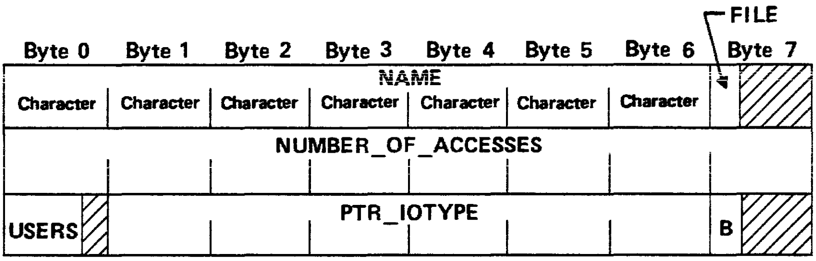


NO1: 86/07/10

The packed record is:

```
TYPE
  table = packed record
    name: string(7),
    file: (bi, di, lg, pr),
    number_of_accesses: integer,
    users: 0 .. 100,
    ptr_iotype: ^iotype,
    b: boolean,
  recend;
```

This record would appear in memory as follows (slashes indicate unused memory):

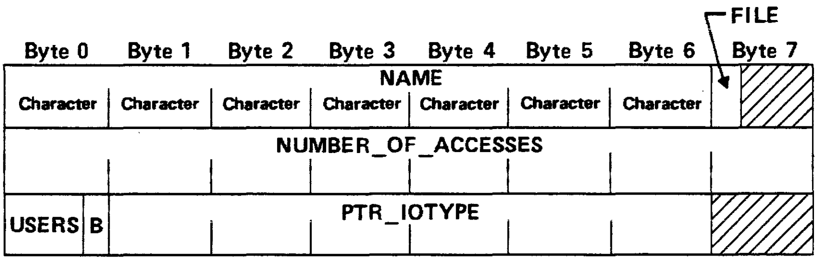


NO2: 86/07/10

The record, as follows, is now rearranged slightly to make more efficient use of the space:

```
TYPE
  table = packed record
    name: string(7),
    file: (bi, di, lg, pr),
    number_of_accesses: integer,
    users: 0 .. 100,
    b: boolean,
    ptr_iotype: ^iotype,
  recend;
```

This record would appear in memory as follows (slashes indicate unused memory):

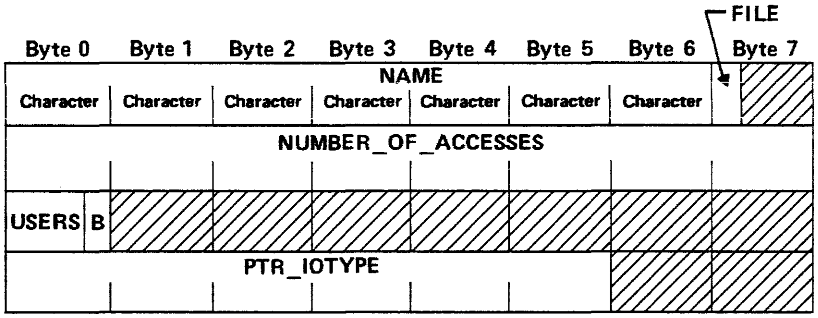


NO3: 86/07/10

The following record declares the pointer field to be aligned at byte zero (the first byte) of a word:

```
TYPE
  table = packed record
    name: string(7),
    file: (bi, di, lg, pr),
    number_of_accesses: integer,
    users: 0 .. 100,
    b: boolean,
    ptr_iotype: ALIGNED [0 MOD 8] ^iotype,
  recend;
```

This record would appear in memory as follows (slashes indicate unused memory):



N04: 86/07/10

# The CYBIL Run-Time Environment F

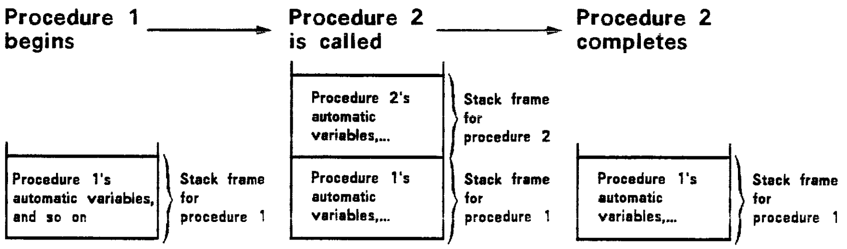
This appendix describes the stack frame mechanism, register assignments during execution, how parameters are passed, external references, and variable allocation.

## Stack Frame Mechanism

Space is allocated when the block in which a variable is declared is entered; space is released when an exit is made from the block. This would be simple if the sequence in which blocks were entered and exited was always linear. But CYBIL allows blocks to be nested within each other and, in the case of a recursive function or procedure, CYBIL allows a block to call itself. Thus, space could be allocated for variables from several procedures at the same time, or for several occurrences of the same variables if a recursive routine is called repeatedly.

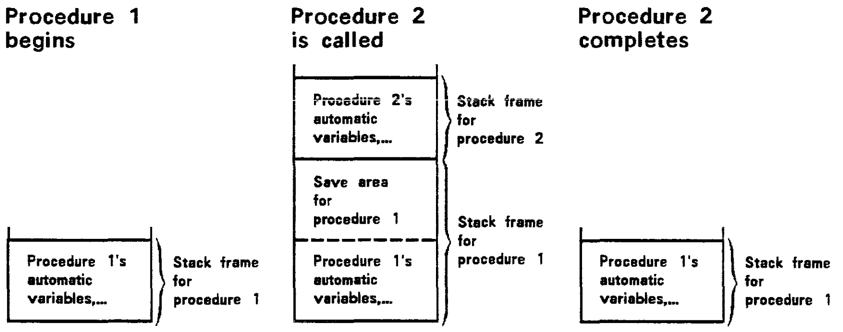
The space allocated for each variable or each occurrence of the same variable must be kept separate to maintain the integrity of the variable throughout execution of the procedures. In addition, to satisfy the scope rules, any procedure that is inside another procedure must be able to access the variables of the outer (enclosing) procedure. However, a procedure must be prevented from accessing the variables of procedures that do not enclose it. The mechanism that allows memory to be allocated following these rules is the stack.

Every executing program has an area of memory called a stack associated with it. The stack expands and contracts as procedures within the program are called and then complete. Each time a procedure is called, an area called a stack frame is allocated in the stack to hold the procedure's local automatic variables and other pertinent information. When the procedure completes, the stack frame for the procedure is released.


Procedure 1 begins
Procedure 2 is called
Procedure 2 completes
Procedure 2's automatic variables,...
Stack frame for procedure 2
Procedure 1's automatic variables, and so on
Stack frame for procedure 1
Procedure 1's automatic variables,...
Stack frame for procedure 1
Procedure 1's automatic variables,...
Stack frame for procedure 1

STK1: 86/08/08

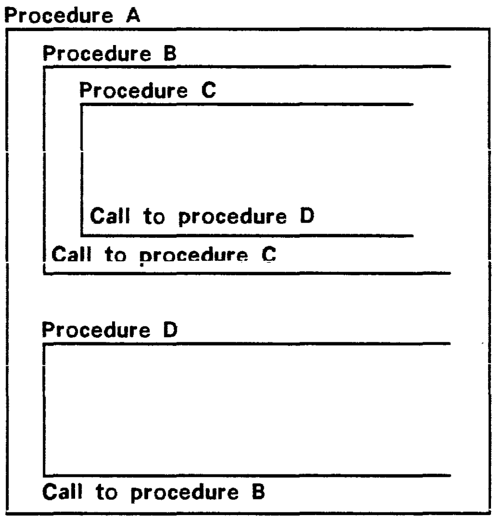
In addition, each time a new procedure is called, the environment of the calling procedure is saved in a part of the stack frame called the save area (or the stack frame save area). This save area contains all the information necessary to resume execution of the calling procedure when the called procedure completes.


Procedure 1 begins
Procedure 1's automatic variables,...
Stack frame for procedure 1
Procedure 2 is called
Procedure 2's automatic variables,...
Stack frame for procedure 2
Save area for procedure 1
Procedure 1's automatic variables,...
Stack frame for procedure 1
Procedure 2 completes
Procedure 1's automatic variables,...
Stack frame for procedure 1

STK2: 86/08/08

Various pointers in the stack frames link calling and called procedures, enclosed and enclosing procedures, the current stack frame and the next space available. (These pointers are controlled by the system, not the program.)

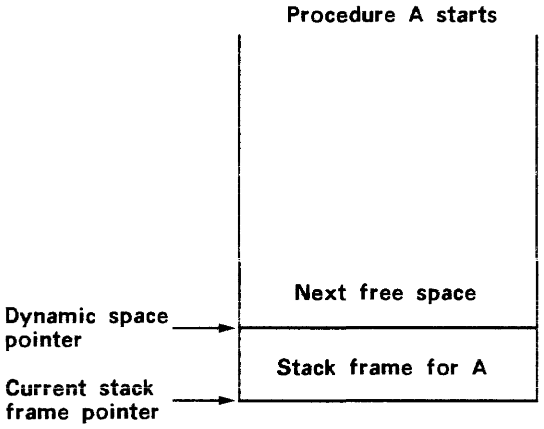
The following example illustrates the process. Consider program A which has two sets of nested procedures: B and C, and D.



PROC1: 86/07/10

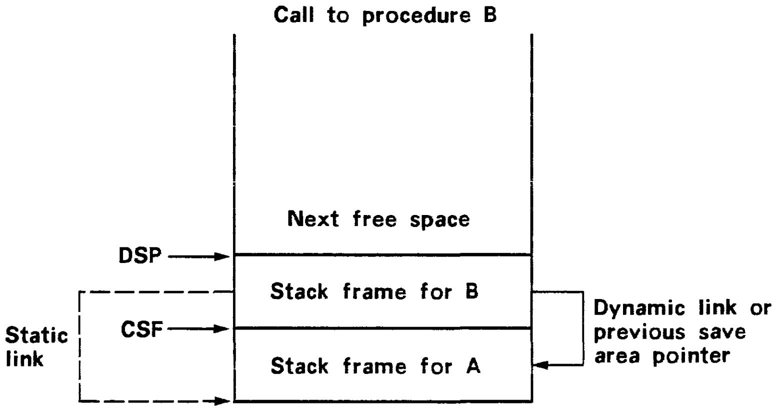
The variables for these procedures are allocated at different times during program execution. Memory for the static variables is allocated at the beginning of the program and remains until the end of the program. Dynamic variables (the local automatic variables within the procedures) are allocated when a procedure is called and released when the procedure completes.

When program A begins, a stack frame for the dynamic variables in program A is created. A pointer called the current stack frame (CSF) points to the beginning of the stack frame and a dynamic space pointer (DSP) points to the next available space in the stack (the top of the current stack frame).



PROCA: 86/07/10

When procedure B is called, A's environment (the contents of registers and so on) is saved in A's stack frame and a new stack frame is created for procedure B.



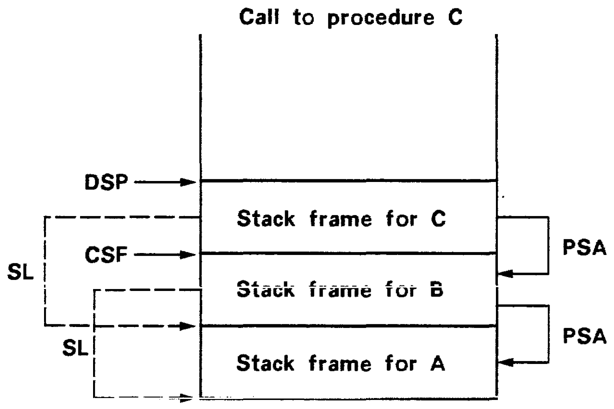
PROCB: 86/07/11

A pointer called a static link (SL) is created that points back to the stack frame for A. A static link always points to the stack frame of the procedure containing the called procedure (and, in this case, A contains B) if the called procedure is an internal procedure of the calling procedure; otherwise, it is meaningless. The static link enables the most recently called procedure to have access to the variables already declared in the procedures that contain it.

A dynamic link pointer is also created that points to A's stack frame. This dynamic pointer is called the previous save area (PSA) pointer, and it points to the area that was saved (that is, the environment) for the previous procedure, A. This pointer allows the environment of the preceding procedure (the calling procedure) to be restored when the current procedure (the called procedure) completes.

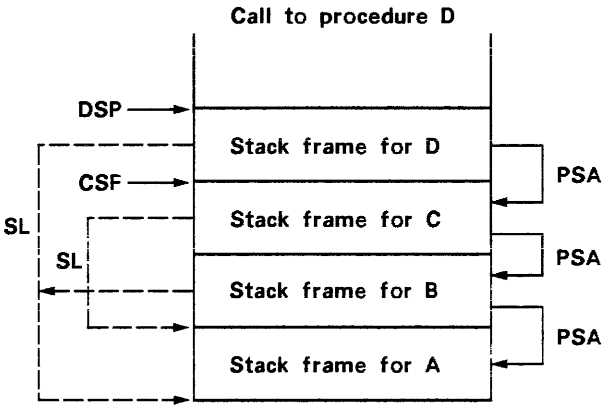
The current stack frame pointer is updated automatically to the start of the current stack frame (the stack frame for procedure B) and the dynamic space pointer moves to the next available space in the stack.

The call to procedure C results in a similar structure. Again the static and dynamic links point to the previous stack frames.



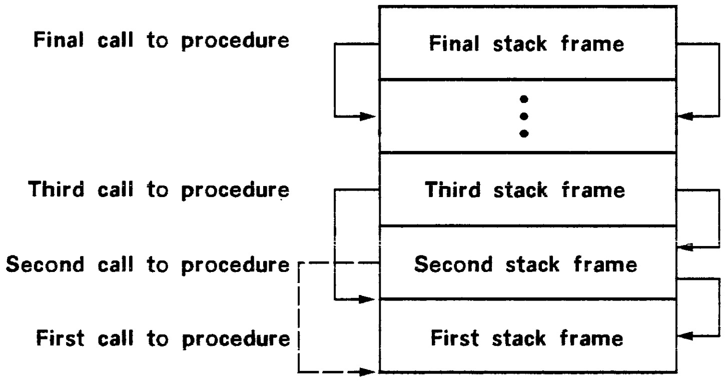
PROCC: 86/07/10

However, when procedure C calls procedure D, the objects of the pointers change. The dynamic link still points to the previous stack frame, but the static link points back to the stack frame for A. This is because procedure D is not contained in blocks B or C, only in block A; therefore, it can only access variables in block A, the outermost level of the program.



PROCD: 86/07/10

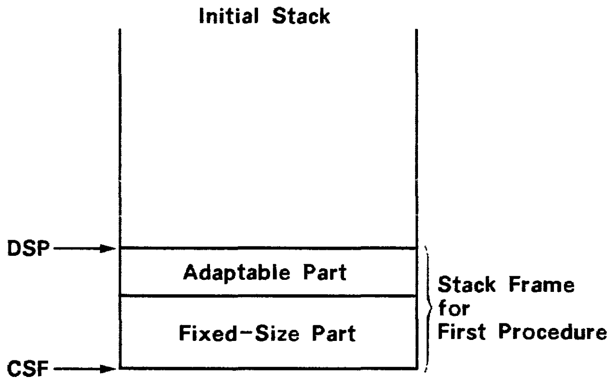
This example shows how a recursive procedure would work. A separate stack frame would be created for each call to the procedure, with the appropriate links backward. Then, as each iteration of the procedure completed, its stack frame would disappear from the stack and the environment saved in the preceding frame would be restored. This would continue until the initial call to the procedure completed.



RECUR: 86/08/15

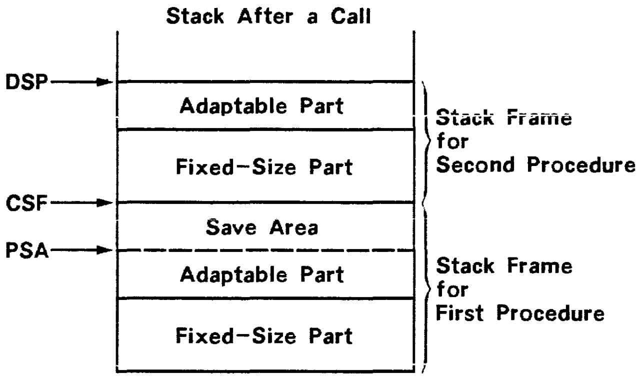
The remainder of this section gives a more detailed description of the format and contents of a stack frame.

The stack frame always consists of at least two parts: a fixed-size part and a variable-size part. The fixed-size part contains all the data whose size is known at compilation time and it includes the automatic variables for the procedure. The variable-size part contains all the adaptable structures whose sizes can be determined only at execution time. In addition, when one procedure calls another, the environment of the first is saved in its stack frame save area. For example, the stack initially appears as follows.



STK3: 86/08/15

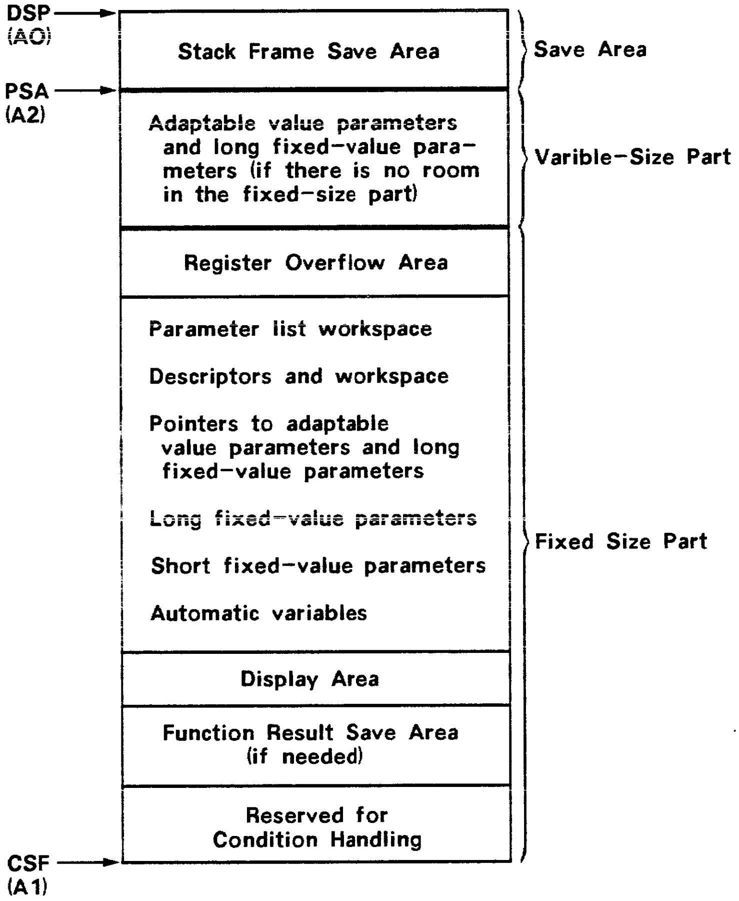
After a call is made to a second procedure, the stack contains two frames as shown below.



STK4: 86/08/15

The stack frame save area exists only after another procedure has been called, not for the current procedure.

A stack frame has the following format.


DSP (A0)
PSA (A2)
CSF (A1)
Stack Frame Save Area
Adaptable value parameters and long fixed-value parameters (if there is no room in the fixed-size part)
Register Overflow Area
Parameter list workspace
Descriptors and workspace
Pointers to adaptable value parameters and long fixed-value parameters
Long fixed-value parameters
Short fixed-value parameters
Automatic variables
Display Area
Function Result Save Area (if needed)
Reserved for Condition Handling
Save Area
Varible-Size Part
Fixed Size Part

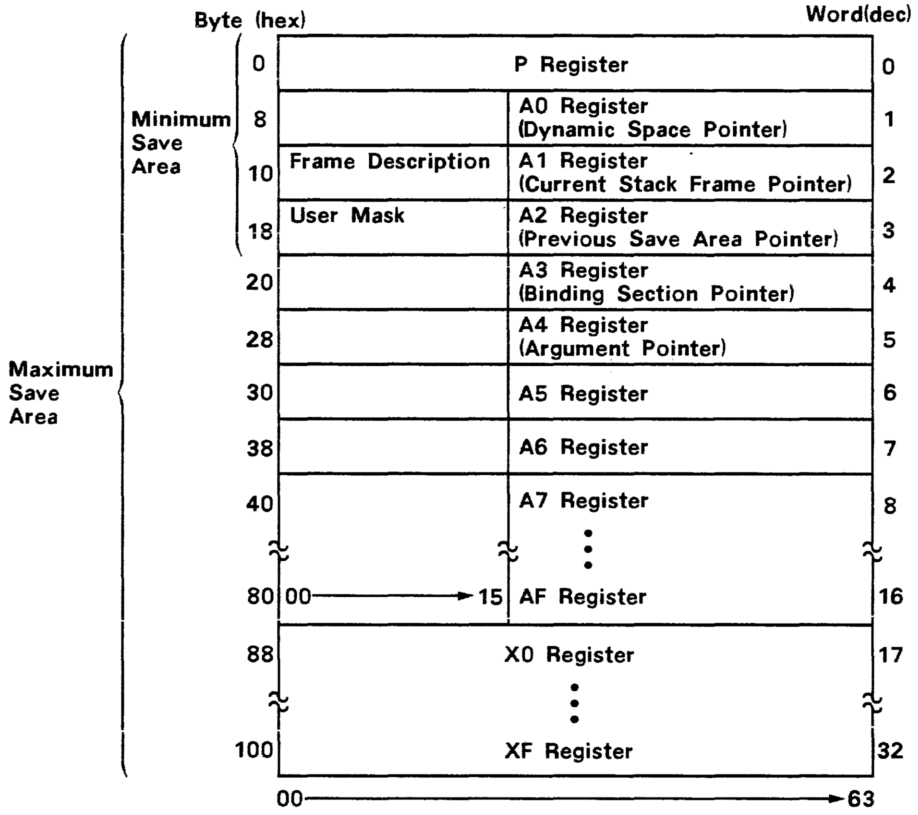
STKFMT: 86/08/15

Each of these areas is discussed next.

## Stack Frame Save Area

The hardware call instruction saves a designated set of registers in the save area, which can be thought of as the top of the rest of the stack frame of the procedure that issued the call. The stack frame of the called procedure is then built above the save area of the calling procedure.

The save area can be broken down into two areas: the minimum save area and the maximum save area.



STKSAV: 86/08/08

The minimum save area contains the P, A0, A1, and A2 registers, the stack frame descriptor, and the user mask. The maximum save area contains everything in the minimum save area plus, optionally, registers A3 through AF and registers X0 through XF. The contents of these registers is given later in this section. The CYBIL System Interface manual contains additional information on the stack frame save area.

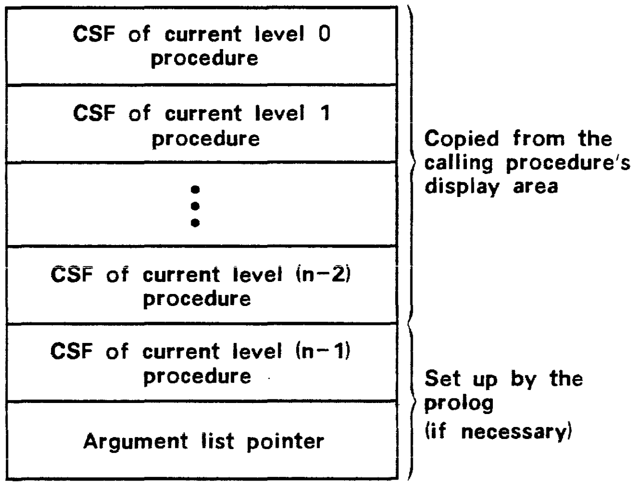
## Fixed-Size Part

The fixed-size part of the stack frame contains:

- An 8-byte, initialized field for condition handling
- A word to be used as the function result save area when a function has a nonlocal exit
- A display area containing the information necessary to access the variables of preceding procedures (the display area is shown in detail below)
- Assorted procedure data (also described following the display area)

As previously mentioned, the display area contains pointers back to the stack frames of all the procedures that contain the current procedure. These pointers make it possible for the current procedure to access the variables of the preceding procedures.

The display area has the following format:


CSF of current level 0 procedure
CSF of current level 1 procedure
•
•
•
CSF of current level (n-2) procedure
CSF of current level (n-1) procedure
Argument list pointer
Copied from the calling procedure's display area
Set up by the prolog (if necessary)

STKDIS: 86/08/08

Each display area entry is a 6-byte pointer (a current stack frame pointer) that is right-justified in its display word. The total size of the display for a specific procedure is based on that procedure's nesting level. The prolog[1] will save the static link (if it was passed in register A5) only if the procedure was nested. The prolog will also save the parameter list pointer (if it was passed in register A4) only if the procedure contains at least one locally defined procedure.

Automatic variables or value parameters may be declared so that all bounds and size information is known at compilation time. In this case, the fixed amount of storage that is required for the variable is allocated from the fixed bound part of the automatic stack.

Adaptable parameters may be declared so that some bounds and size information is not known at compilation time. In this case, a type descriptor must be allocated for the type, containing the result of the calculation of all variable bounds and a variable descriptor that can locate the base address of the variable bound part of the automatic stack. These descriptors are allocated in the fixed bound part of the automatic stack. In addition, a workspace may be required in the fixed-size part to hold intermediate results for execution-time descriptor calculations.

1. A prolog is a set of one or more instructions that is executed at the beginning of every procedure and function to set up the registers properly for that procedure or function.

The parameter lists for procedure calls are held in a fixed-size area. If the current procedure calls other procedures, the parameter list must be allocated in its own fixed part area. Each actual parameter is represented in the parameter list as either a value or a pointer. If the parameter is passed by value and its formal parameter length is less than or equal to 8 bytes, the parameter is represented in the list by its value in the smallest number of bytes required to hold the value. All other parameters are represented by 6-byte pointers (plus a descriptor if necessary). Further information on passing parameters is given later in this section.

The register overflow area contains hardware registers that are preempted during execution. The size of this workspace can be determined at compilation time.

## Variable-Size Part

The variable-size part of the stack frame contains storage for all adaptable value parameters whose bounds and size information cannot be determined at compilation time. The descriptors for these variables are contained in the fixed-size part.

## Register Assignments

While a CYBIL program is executing, the following registers are assigned:

| Register (Hexadecimal) | Contents |
|---|---|
| A0 | Dynamic space pointer (DSP) |
| A1 | Current stack frame pointer (CSF) |
| A2 | Previous save area pointer (PSA) |
| A3 | Binding section pointer (BSP) |
| A4 | Argument list pointer (ALP) |
| A5 | Static link (SL) |
| AA through AE and X9 through XD | Parameters passed to internal procedures or functions |
| X0 | Number of parameters passed (in bits 40 through 43) |
| XE | Line number for range checking (LN) |
| AF | Function result (if it is a simple pointer) |
| XF | Function result (if it is scalar) |

The dynamic space pointer, current stack frame pointer, previous save area pointer, and static link are described earlier. The binding section pointer indicates the binding section of the procedure that is currently executing. The argument list pointer points to the parameter list passed by the calling procedure.

Registers A0, A1, and A2 always contain the values shown previously. Registers A5, AA through AF, and X9 through XF may be assigned other values during execution of the procedure.

External procedures use registers A1, A3, and A4. Internal procedures use all of these registers in addition to A5. A function can be thought of as a procedure that returns a value. Therefore, the register conventions for function references are the same as those for procedure references.

For internal procedure and function references (that is, those that have neither the XREF nor XDCL attribute declared), the parameters that fit in an A or X register are passed in registers AA through AE and X9 through XD. The A registers are used to pass pointers and the X registers are used to pass the other basic types that fit in a register. The registers are filled starting with AA and X9 respectively, and the parameters are loaded starting from the left of the actual parameter list. If all the required registers are already in use passing other parameters, the parameter yet to be loaded is included with the normal parameter list entries.

The result returned by a function reference is in registers or memory, depending on the type of value being returned. If the function result is a simple pointer, the value is returned as a PVA in register AF. If the function result is a scalar of length less than or equal to 64 bits, it is returned aligned on the right in the XF register. Any unused bits are filled with zero. If the function result is not one of these types, it is stored left-justified as the first element of the parameter list. The second element of the parameter list, in this case, specifies the first actual parameter.

## How Parameters are Passed

The following paragraphs describe how parameters are passed for reference parameters and value parameters, and calls between languages.

### Reference Parameters

For reference parameters, a pointer to the actual data is generated and the pointer is passed as the parameter. The parameter is left-aligned on a word boundary.

### Value Parameters

If the parameter length is less than or equal to a word, a copy of the actual parameter is made in the parameter list. The parameter is right-aligned, but on a word boundary.

If the value parameter is larger than one word in length, the parameter list contains a pointer to the actual parameter or a copy of the actual parameter (this pointer is left-aligned on a word boundary).

Normally, a copy is made of the data. However, the value parameter is not copied to the caller's stack and, instead, a pointer to the data is put in the parameter list under the following conditions (these are the rules for copying large value parameters in the calling procedure):

- A copy is not made of a value parameter that is passed as a value parameter to another procedure
- A copy is not made of a variable that is defined as part of a memory section with the READ attribute specified
- A copy is not made of a large constant that is passed as a value parameter
- A copy is not made of a parameter that will be copied in the called procedure
- A copy is not made of a large (that is, larger than one word) automatic variable that is passed as a value parameter, except under one of the following conditions:
    - The automatic variable is passed by reference in any procedure call within the scope of the defining procedure;
    - A pointer is generated to the automatic variable in an assignment statement using the pointer symbol (^) or the #LOC function, or the ^ symbol or #LOC function is used to pass the address of the automatic variable in the procedure call statement;
    - The automatic variable is being passed to a nested procedure within the scope of the procedure that defined the automatic variable, and the automatic variable is modified in a nested procedure within the scope of the procedure that defined the automatic variable;
    - The automatic variable is a pointer and the pointer is dereferenced on the procedure call.

Value parameters in the prolog of the called procedure are copied if the called or nested procedure generates a pointer to the value parameter via the pointer symbol (^) or the #LOC function and one of the following conditions is true:

- The pointer is passed as a parameter
- The pointer is not an automatic variable of the procedure
- The pointer is assigned to another pointer
- A pointer to the pointer is generated in an assignment statement using the ^ symbol or the #LOC function
- A pointer to the pointer is passed in a procedure call using the ^ symbol or the #LOC function
- The pointer is dereferenced on the left side of an assignment statement
- The pointer is dereferenced on a procedure call and the call is by reference
- The value parameter is a sequence or structure that contains a sequence, the pointer is generated to the sequence, a data item pointer is generated into the sequence with a NEXT statement, and the data item pointer escapes (that is, one of the items previously listed occurs)

If the called procedure or a contained procedure generates a pointer to the value parameter and the value of that pointer escapes or the object of the pointer is altered, the called procedure's prolog copies the parameter to its stack frame. The prolog also generates a pointer to the copied data and stores it in the called procedure's stack. Generation of the pointer to the parameter is performed because the calling procedure may be executing in a different ring than the called procedure.

## Calls Between Languages

If a call could be made to another language and it has a system format actual parameter list to be passed that contains only reference parameters, the parameter list is immediately preceded by a word whose value is the 64-bit integer zero. This word need not precede any other system format actual parameter lists, only those for calls to another language.

## External References

During the compilation process, a hash is computed for each variable and procedure that is externally declared or referenced. This hash is based on an accumulation of the data typing.For procedures, the parameter list (the name of the type of each formal parameter) is included in the process. Before execution, the loader checks these hash values to ensure that the data types for all externally declared and referenced variables agree. If they do not agree, the loader generates an error message documenting the parameter verification errors (the condition identifier is LLE$DECLARATION_MISMATCH and the condition code is LL 161). This means that if you compile modules at different times and the interfaces change between these times, the loader checks and lets you know.

## Variable Allocation

Space for variables is allocated in the order in which they occur in the input stream; no reordering is done. If a variable is not referenced, no space is reserved.

The components of unpacked arrays and records are mapped to memory in their natural order; that is, if an array or record was placed in a sequence and the sequence was reset to the array or record, the following statements would be true:

- A NEXT statement performed on a pointer to the appropriate component type would yield a pointer to the first element of the array or field of the record

- Subsequent NEXT statements performed on pointers of appropriate component types would yield pointers to the second, third, and so forth elements of the array or fields of the record

If an array or record placed in a sequence is retrieved component by component, the types of the data accessed must match the types of the corresponding elements of the array or fields of the record.

# Programming Recommendations H

This appendix lists ways you can increase the efficiency of your source code and improve compilation and loading time. As with any programming recommendations, they should be used only when they do not affect the clarity of your code.

If you would like to share other methods you have found that improve performance, please send them to us using the comment sheet in the back of the manual or send them to the address shown under Submitting Comments in the preface.

## Increasing Source Code Efficiency

The following are suggestions for improving source code efficiency.

- Turn off all range checking (it requires additional storage and is time-consuming) or code your source program to avoid range checking as much as possible. To turn off range checking, specify RUNTIME_CHECKS=NONE on the CYBIL command, or include the directive ?? SET(CHKRNG := OFF) ?? in the source program itself. However, setting RUNTIME_CHECKS to NONE while you are debugging your program is not recommended because legitimate program errors may not be diagnosed.

  It is preferable, therefore, to request range checking but code your program to minimize the amount of code that must be generated to do so, using subranges of types rather than the entire type itself. The following example illustrates this.

  ```
  TYPE
    a_range = 0 ..10;
  VAR
    index, y: a_range,
    x: array [a_range] of integer;
           :
  y := 5;
  index := y;
  x[index] := 3;
  ```

This example declares a subrange from 0 through 10 to be a type named A_RANGE. The variables INDEX and Y are of type A_RANGE, the subrange. Therefore, the assignment statement

```
index := y;
```

is not checked for range violations even if range checking has been requested. Likewise, the assignment statement

```
x[index] := 3;
```

is not checked. If, however, the variables INDEX and Y had been declared integer or some other type besides a subrange, range checking code would have been required.

- Turn off all checking options when you compile the code to improve execution time.
- If you call a procedure repeatedly within a loop structure, call the procedure once and put the loop tests inside the procedure to avoid significant overhead.
- Observe that a procedure should reference only static variables, arguments, and its own automatic variables to avoid overhead associated with those references via the static link.
- If a comparison of two records occurs, organize the fields within the records so that the fields most likely to differ appear first.
- Move a single structure of elements instead of many individual elements. This may require arranging the elements specifically for this purpose (for example, within a record).
- Reference a fixed-size structure rather than an adaptable structure, because the adaptable structure has a descriptor field that must be accessed first and therefore takes more time.
- Reference fields within a record, as it causes no extra execution time.
- If you are repeatedly referencing a complex data structure (by pointers or an indexing process), use a local pointer to access the structure and replace the more complex references.

- Avoid inappropriate use of a null string (for example, moving a null string to a null string) which can result in a no-op instruction that wastes execution time.

- Initialize static variables at compile time, as it causes no overhead at execution time.

- If you initialize a record with constants at execution time, define a static, initialized variable of the same record type and assign that record rather than the individual fields.

- Use a packed structure when conserving space is more important than access time. Generally, a packed structure requires less storage than an unpacked structure. However, be aware there may be greater overhead in accessing the structure's components because the elements of a packed structure may not lie on addressable boundaries.

- When you organize data within a packed structure, group the elements together that are aligned by bit.

- Declare a string type rather than a packed array of characters.

- Use these data structures for homogeneous data: the array, the sequence, and the heap, in order of their effectiveness.

- Use sequences rather than heaps, as they are more efficient in terms of storage and execution time.

- Use the NEXT and RESET statements (used on sequences and user heaps) which are implemented as inline code and are faster than the ALLOCATE and FREE statements.

- Avoid allocating small types in a heap (using the ALLOCATE statement), which causes a large overhead. Space is allocated only when the ALLOCATE statement is executed but, each time it is executed, a header is also added to maintain chaining information.

- Use the PUSH statement (rather than combining the ALLOCATE and FREE statements), as it is implemented as inline code and is, therefore, much faster.

- When specifying a pointer, use the pointer symbol ^ rather than the #LOC function, to promote efficiency and maintainability.

- If the definition of a structure contains many flags or attributes, consider the following when choosing between a boolean type and a set type:

  - If the record is unpacked, using the set type reduces the size of the definition.

  - Any subset of the attributes of a set can be tested immediately.

  - If you are testing a single element, an unpacked boolean type is more efficient than a set type.

- Use boolean expressions rather than conditional statements, as they are more efficient. For example, the statement

  ```
  equality := (a = b);
  ```

  is more efficient than the IF statement

  ```
  IF a = b THEN
    equality := TRUE;
  ELSE
    equality := FALSE;
  IFEND;
  ```

- When possible, use a CASE statement rather than a long, complex IF sequence. This can be done when the value of a single variable determines the sequence of action.

- Arrange compound boolean expressions so that the first condition evaluated is the one most likely to end evaluation of the entire expression.

- Group constants at the same level in an expression together. For example, the expression

  ```
  x := 5 * y * z * 2;
  ```

  produces object code using two constants (5 and 2) and two variables (Y and Z). However, if the expression were rewritten as

  ```
  x := 5 * 2 * y * z;
  ```

  with the constants together, the compiler would evaluate the expression 5 * 2 to get the constant 10, then produce object code using only one constant (10) and two variables (Y and Z).

- Define positive integer subranges. When dividing by a power of 2 on a positive integer subrange, a shift instruction can be generated. A shift instruction is considerably faster than a divide instruction.

- To concatenate substrings, use substring assignment rather than the STRINGREP procedure wherever possible. The STRINGREP procedure is implemented as a call to an execution-time library routine and it is inefficient compared to substring assignment.

- Do not open a file before it is needed and close it when it is no longer needed. Unnecessary openings and closings cause some overhead.

- When using an adaptable string, specify the length parameter wherever possible to provide the compiler with the maximum length.

- Avoid references to variables declared with the XDCL attribute and within a section as they are made via the binding section and, therefore, cause some overhead.

- Make the initial access to a variable that is used within a loop outside the loop, as the code generator does not move invariant code out of the loop.

- Use the ANALYZE_PROGRAM_DYNAMICS command and the Measure Program Execution Utility to study your program's efficiency with respect to execution time, page faults, and module connectivity. Both are described in the SCL Object Code Management manual.For more detailed data collection and reporting, use the ACTIVATE_JOB_STATISTICS command or, if you have the required permission, the ACTIVATE_SYSTEM_STATISTICS command, and the Display Binary Log Utility (refer to the System Performance and Maintenance manual).

- Bind programs to improve overall load and execution time. You can bind programs with the ANALYZE_PROGRAM_DYNAMICS command, the Measure Program Execution Utility, or either of the Object Library Generator subcommands CREATE_MODULE or BIND_MODULE. All of these commands and utilities are described in the SCL Object Code Management manual.

- Create a linked module for large programs with a great deal of static data. For further information on linked modules, refer to the SCL Object Code Management manual.

- Organize the frequently used variables first in large user stacks to avoid reaching the threshold of the load and store instructions ($2^{16}$), causing an extra instruction to be generated to handle the offset.

- Check the default values for compilation options and use the appropriate values. To improve performance, select optimization, if possible. Unless you are debugging code, avoid selecting stylized code for debugging and range checking; both generate extra code and cause greater execution time.

## Improving Compilation and Loading Time

The following are suggestions for improving compilation and loading time.

- To improve compilation time, avoid selecting the following options on the CYBIL command:
  - Generating debug tables for the Debug Utility
  - Generating stylized code
  - Generating range-checking code
  - Selecting listings (including the source listing, the cross-reference listing, the attribute listing, and the object code listing)
  - Generating a source listing that includes the generated code rather than generating the source listing alone
- Bind programs to improve overall loading time. You can bind programs with the ANALYZE_PROGRAM_DYNAMICS command, the Measure Program Execution Utility, or either of the Object Library Generator subcommands CREATE_MODULE or BIND_MODULE. All of these commands and utilities are described in the SCL Object Code Management manual.
- Avoid selecting Debug tables (the symbol table and line table) as they slow the loading process. They should not be selected during compilation unless debugging is necessary.

# Differences Between CYBIL and Pascal I

The CYBIL language is similar to Pascal. This appendix lists the differences between CYBIL and Pascal for NOS/VE.

The following features are available in CYBIL but not Pascal:

- XDCL and XREF attributes for procedures, functions, and data.
- The STATIC attribute for data.
- MODULE and MODEND keywords to delimit compilation units.
- Strings and substrings with size coercions, and coercions between the $CHAR function and single-parameter substrings.
- Constant expressions.
- Indefinite value constructors for initialization.
- Initialization of static data.
- Adaptable arrays (with numeric index type and fixed item size only) with upper and lower bound functions, adaptable strings, and allocation designators.
- Enhanced memory management facilities using sequences and user heaps, and the PUSH, NEXT, RESET, ALLOCATE, and FREE statements.
- Fixed and adaptable sequences and user heaps.
- Group labels. Labels are optional before the BEGIN, FOR, WHILE, and REPEAT statements and after the END, FOREND, and WHILEND statements. When used before and after the statement groups BEGIN/END, FOR/FOREND, and WHILE/WHILEND, the names specified for the labels must match. Labels are required after the EXIT and CYCLE statements. The name specified for a label must be unique among all names known by the containing procedure. The scope of a group label is the group statement it precedes; it does not extend to procedures called from within its scope.
- The grouping delimiters PROCEND, RECEND, FOREND, WHILEND, IFEND, and CASEND, which replace the Pascal keyword END or other devices used to signal the end of such structures.

- Use of equality (=) and inequality (<>) relationships on records.
- Subranges and the ELSE option as CASE statement selections.
- The CELL and pointer to cell types, and the #LOC and #SIZE functions.
- Data mapping as described in appendix E, Data Representation in Memory.
- The evaluation of condition terms proceeding from left to right and ending when the value of the condition is determined.
- Pointer to procedure and pointer to function type.
- The relative pointer type.
- Special characters _, @, #, and $ permitted in names.
- Comments terminated by an end of line.
- Multiple instances of CONST, TYPE, VAR, and procedure declarations permitted in any order.
- The reserved word RETURN as a control statement.
- EXIT and CYCLE followed by a group label as control statements.
- CASE statement selectors delimited on both sides by the equal sign character (=) rather than only on the end by the colon character (as in Pascal).
- Alignment of data on specific boundaries.
- Control over data locality using the SECTION declaration.
- Restricted ability to generate certain inline instructions using intrinsics.
- The ELSEIF option associated with the IF statement.
- Compilation time facilities.
- Hexadecimal, octal, and binary numeric constants.
- Detection of side effects from programmer-defined functions.
- The bound variant record type.

- Directives to control the compilation process.
- INLINE procedures and functions.

The following are not used in CYBIL:

- Procedures and functions as parameters.
- Forward procedure declarations.
- The FILE type, I/O functions, and predicates.
- Label declarations.
- The GO TO statement.
- The WITH statement.
- PACK and UNPACK operations.
- Set value constructors in variable expressions.
- String concatenation, variable strings, and the functions INDEX and LENGTH.
- Alternate forms of multidimensioned arrays. CYBIL allows multidimensioned arrays to be declared only in the form

  ```
  ARRAY [index1] OF ARRAY [index2] OF ... }
  ```

  whereas Pascal also allows the form

  ```
  ARRAY [index1, index2,...] }
  ```

- PACKED SET types.
- The built-in functions ABS, SQR, SIN, COS, EXP, LN, SQRT, ARCTAN, ODD, EOF, EOLN, CARD, CLOCK, and ROUND.
- The built-in procedures DATE, DISPOSE, GET, HALT, MESSAGE, NEW, PAGE, PACK, PUT, READ, READLN, RESET, REWRITE, TIME, UNPACK, WRITE, and WRITELN.
- A predefined MAXINT constant [however, you can achieve the same result in CYBIL using the UPPERVALUE(INTEGER) function].

The implementation of the following items differs:

- The CASE statement syntax.
- Designation of the end of declarations and statements (use of a semicolon rather than an end of line).
- Designation of comments.
- Externally referenced variables, functions, and procedures (use of the XREF attribute rather than EXTERNAL).
- Names of certain functions ($CHAR instead of CHR, $INTEGER instead of ORD, and STRLENGTH instead of MAXLENGTH).
- Reserved (predeclared) names as listed in appendix D, Reserved Words.

# Constant and Type Declarations for CYBIL I/O J

This appendix lists the constant and type declarations used by the CYBIL I/O procedures described in this manual. The declarations are listed in alphabetical order by identifier name.

## Constants

```
cyc$ada                      = 'ADA                           ',
cyc$apl                      = 'APL                           ',
cyc$assembler                = 'ASSEMBLER                     ',
cyc$basic                    = 'BASIC                         ',
cyc$c                        = 'C                             ',
cyc$cobol                    = 'COBOL                         ',
cyc$cybil                    = 'CYBIL                         ',
cyc$debugger                 = 'DEBUGGER                      ',
cyc$fortran                  = 'FORTRAN                       ',
cyc$lisp                     = 'LISP                          ',
cyc$pascal                   = 'PASCAL                        ',
cyc$pli                      = 'PLI                           ',
cyc$ppu_assembler            = 'PPU_ASSEMBLER                 ',
cyc$prolog                   = 'PROLOG                        ',
cyc$scl                      = 'SCL                           ',
cyc$scu                      = 'SCU                           ',
cyc$unknown_processor        = 'UNKNOWN                       ',
cyc$vx                       = 'VX                            ';

cyc$ascii_log                = 'ASCII_LOG                     ',
cyc$binary                   = 'BINARY                        ',
cyc$binary_log               = 'BINARY_LOG                    ',
cyc$data                     = 'DATA                          ',
cyc$file_backup              = 'FILE_BACKUP                   ',
cyc$legible                  = 'LEGIBLE                       ',
cyc$legible_data             = 'LEGIBLE_DATA                  ',
cyc$legible_library          = 'LEGIBLE_LIBRARY               ',
cyc$legible_unknown          = 'LEGIBLE_UNKNOWN               ',
cyc$list                     = 'LIST                          ',
cyc$list_unknown             = 'LIST_UNKNOWN                  ',
cyc$object                   = 'OBJECT                        ',
cyc$object_data              = 'OBJECT_DATA                   ',
cyc$object_library           = 'OBJECT_LIBRARY                ',
cyc$screen                   = 'SCREEN                        ',
cyc$screen_form              = 'SCREEN_FORM                   ',
cyc$unknown_contents       = 'UNKNOWN                        ';

cyc$detach_file = cyc$return_file;

cyc$max_file_name_size = 512;

cyc$min_ecc = (($INTEGER('C')*100(16))+$INTEGER('Y'))
  *1000000(16);
cyc$max_ecc = cyc$min_ecc + 9999;

cyc$min_ecc_cybil_input_output  = cyc$min_ecc + 6200;
cyc$max_ecc_cybil_input_output  =
  cyc$min_ecc_cybil_input_output + 99;

cyc$page_limit = 439804651103;

cyc$title_size = 45;

cyc$wide_page_width = 132;
cyc$narrow_page_width = 80;
cyc$max_page_width = 65535;
```

## Types

```
cyt$close_file_disposition = (cyc$delete_file, cyc$retain_file,
  cyc$return_file, cyc$unload_file,
  cyc$default_file_disposition);

cyt$current_file_position = (cyc$beginning_of_information,
  cyc$middle_of_record, cyc$end_of_record, cyc$end_of_block,
  cyc$end_of_partition, cyc$end_of_information);

cyt$file = ^SEQ ( * );

cyt$file_access = (cyc$read, cyc$write, cyc$read_write);

cyt$file_character_set = (cyc$ascii, cyc$ascii612, cyc$ascii812,
      cyc$display_64, cyc$reserved_code1, cyc$reserved_code2);

cyt$file_contents = string (31);

cyt$file_existence = (cyc$new_file, cyc$old_file,
  cyc$new_or_old_file);

cyt$file_kind = (cyc$binary_file, cyc$display_file,
  cyc$record_file, cyc$text_file);

cyt$file_name = string ( * <= cyc$max_file_name_size);

cyt$file_processor = string (31);

cyt$file_specification = record
  case selector: cyt$file_specification_selector of
  = cyc$file_kind =
    file_kind: cyt$file_kind,
  = cyc$file_access =
    file_access: cyt$file_access,
  = cyc$file_existence =
    file_existence: cyt$file_existence,
  = cyc$open_position =
    open_position: cyt$open_close_position,
  = cyc$close_file_disposition =
    close_disposition: cyt$close_file_disposition,
  = cyc$file_contents =
    file_contents: cyt$file_contents,
  = cyc$file_processor =
    file_processor: cyt$file_processor,
  = cyc$file_character_set =
    file_character_set: cyt$file_character_set,
  = cyc$new_page_procedure =
    new_page_procedure: cyt$new_page_procedure,
  = cyc$page_length =
    page_length: cyt$page_length,
  = cyc$page_width =
    page_width: cyt$page_width,
  = cyc$page_format =
    page_format: cyt$page_format,
  = cyc$future_spec1 =
    ,
  = cyc$future_spec2 =
    ,
  = cyc$future_spec3 =
    ,
  = cyc$future_spec4 =
    ,
  = cyc$future_spec5 =
    ,
  casend,
recend;
```

```
cyt$file_specifications = ^array [1 .. * ] of
  cyt$file_specification;

cyt$file_specification_selector = (cyc$file_kind,
  cyc$file_access, cyc$file_existence, cyc$open_position,
  cyc$close_file_disposition, cyc$file_contents,
  cyc$file_processor, cyc$file_character_set,
  cyc$new_page_procedure, cyc$page_length, cyc$page_width,
  cyc$page_format),

cyt$new_page_procedure = record
  case kind: cyt$page_procedure_kind of
  = cyc$user_specified_procedure =
    user_procedure: cyt$user_page_procedure,
  = cyc$standard_procedure =
    title: string (cyc$title_size),
  = cyc$omit_page_procedure =
    ,
  casend,
recend;

cyt$open_close_position = (cyc$beginning, cyc$end, cyc$asis,
      cyc$default_open_position);

cyt$page_format = (cyc$continuous_form, cyc$burstable_form,
      cyc$non_burstable_form, cyc$untitled_form);

cyt$page_length = 1 .. cyc$page_limit;

cyt$page_width = 1 .. cyc$max_page_width;

cyt$page_procedure_kind = (cyc$user_specified_procedure,
  cyc$standard_procedure, cyc$omit_page_procedure);

cyt$skip_direction = (cyc$forward, cyc$backward);

cyt$skip_unit = (cyc$record, cyc$block, cyc$partition);

cyt$system_type = (cyc$nosve, cyc$nos, cyc$nosbe, cyc$vsos,
  cyc$eos, cyc$aegis);

cyt$user_page_procedure = ^procedure (display_file: cyt$file;
  next_page_number: integer;
  VAR status: ost$status);
```

```
ost$status = record
  case normal: boolean of
  = FALSE =
    condition: ost$status_condition_code,
    text: ost$string,
  = TRUE =
    ,
  casend,
recend;
```

# Messages for CYBIL I/O K

This section describes the status messages that may result from either the improper use of CYBIL I/O or from detection of an error. If one of these conditions arises, the status condition will be returned in the status variable parameter. The structure of the status record returned by this variable is described in chapter 9, How to Use CYBIL I/O.

In the message descriptions listed below, filename will be replaced by the name of a specific file when the message appears in the message template.

| Message | Meaning |
|---|---|
| FILE NAME TOO LONG, filename | The specified file name has more characters than the operating system will allow. |
| FILE NOT OPEN | An undefined variable of type cyt$file was passed to a CYBIL I/O procedure other than one of the open procedures. The file name is not known. |
| INCORRECT FILE NAME, filename | An attempt was made to open a file with a name that does not conform to the file naming conventions of the operating system. |
| INCORRECT INPUT REQUEST FOR FILE filename | An attempt was made to read from a file that was opened only for output. |
| INCORRECT DISPLAY LINE POSITION FOR FILE filename | A line number less than 1 was passed to the CYP$POSITION_DISPLAY_PAGE procedure. |
| INCORRECT TAB COLUMN FOR FILE filename | A tab_column less than 1 was passed to the CYP$TAB_FILE procedure. |
| INCORRECT OPEN REQUEST FOR FILE filename | An invalid combination of parameters was given to an open procedure (for example, "CYC$NEW_FILE, CYC$READ" is incorrect). |

| | |
|---|---|
| INVALID OPERATION ATTEMPTED ON FILE filename | An operation was attempted that does not match the FILE_KIND specified for the file on the call to the open file procedure. (For example, a CYP$GET_NEXT_BINARY may have been attempted on a file opened as a text file.) |
| INCORRECT OUTPUT REQUEST FOR FILE filename | An attempt was made to write to a file that was opened only for input. |
| INCORRECT SKIP COUNT filename | A skip count less than -1 was passed to the CYP$SKIP_LINES procedure. |
| KEY BEYOND E-O-I ON FILE filename | An attempt was made to perform a binary file operation with a key that was outside the bounds of the file (in other words, the key did not specify a random address that is in the file). |
| PREMATURE END OF OPERATION ON FILE filename | A boundary condition was encountered during the CYP$POSITION_ RECORD_FILE procedure before the count was exhausted. |
| NO MEMORY TO OPEN FILE filename | There was insufficient space to allocate the descriptor and/or buffer for the file. |
| COULD NOT FIND FILE filename | An attempt was made to open an old file that CYBIL I/O cannot find. |
| FILE filename ALREADY EXISTS | An attempt was made to open a new file but a file with that name already exists. |

# Index

# Index

Comments (continued from other side)

'lease fold on dotted line;
eal edges with tape only.

FOL

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

**CONTROL DATA**
**Technology & Publications Division**
**ARH219**
**4201 N. Lexington Avenue**
**Arden Hills, MN** 55126-6198

# CYBIL Language Definition 60464113 F

We value your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

| **Who are you?** | **How do you use this manual?** |
|---|---|
| ☐ Manager | ☐ As an overview |
| ☐ Systems analyst or programmer | ☐ To learn the product or system |
| ☐ Applications programmer | ☐ For comprehensive reference |
| ☐ Operator | ☐ For quick look-up |
| ☐ Other ______________ | |

What programming languages do you use? ______________

**How do you like this manual?** Check those questions that apply.

| Yes | Somewhat | No | |
|---|---|---|---|
| ☐ | ☐ | ☐ | Is the manual easy to read (print size, page layout, and so on)? |
| ☐ | ☐ | ☐ | Is it easy to understand? |
| ☐ | ☐ | ☐ | Does it tell you what you need to know about the topic? |
| ☐ | ☐ | ☐ | Is the order of topics logical? |
| ☐ | ☐ | ☐ | Are there enough examples? |
| ☐ | ☐ | ☐ | Are the examples helpful? (☐ Too simple? ☐ Too complex?) |
| ☐ | ☐ | ☐ | Is the technical information accurate? |
| ☐ | ☐ | ☐ | Can you easily find what you want? |
| ☐ | ☐ | ☐ | Do the illustrations help you? |

**Comments?** If applicable, note page and paragraph. Use other side if needed.

**Would you like a reply?** ☐ Yes ☐ No

From:

| | |
|---|---|
| Name | Company |
| Address | Date |
| | Phone |

Please send program listing and output if applicable to your comment.

## Keyword Index

## Statement Index

## Function Index

## Procedure Index

## Formatting and Compilation Index

## CYBIL I/O Procedure Index